NVIDIA OpenGL Extension Specifications

February 22, 2008

This document is an abridged collection of OpenGL extension specifications limited to those extensions for *new* OpenGL functionality introduced by the GeForce 8 Series (G8*x*) architecture. See the unabridged document "NVIDIA OpenGL Extension Specifications" for a complete collection.

NVIDIA-specific OpenGL extension specifications, possibly more up-to-date, can be found at:

http://developer.nvidia.com/view.asp?IO=nvidia_opengl_specs

Other OpenGL extension specifications can be found at:

http://oss.sgi.com/projects/ogl-sample/registry/

Corrections? Email opengl-specs@nvidia.com

Table of Contents

Table of NVIDIA OpenGL Extension Support

Extension	NV1*x*	NV2*x*	*NV3x*	*NV4x*	*G8x*	Notes
ARB_color_buffer_float				R75	X	
ARB_depth_texture		R25+	X	X	X	1.4 functionality
ARB_draw_buffers				R75	X	2.0 functionality
ARB_fragment_program			X	X	X	
ARB_fragment_program_shadow			R55	X	X	
ARB_fragment_shader			R60	X	X	2.0 functionality, GLSL
ARB_half_float_pixel			R75	R75	X	
ARB_imaging	R10	X	X	X	X	1.2 imaging subset
ARB_multisample		X	X	X	X	1.3 functionality
ARB_multitexture	X	X	X	X	X	1.3 functionality
ARB_occlusion_query		R50	R50	R50	X	1.5 functionality
ARB_pixel_buffer_object	R80	R80	R80	R80	X	2.1 functionality
ARB_point_parameters	R35	R35	X	X	X	1.4 functionality
ARB_point_sprite	R50	R50	R50	X	X	
ARB_shader_objects	R60	R60	R60	X	X	2.0 functionality, GLSL
ARB_shading_language_100	R60	R60	R60	X	X	2.0 functionality, GLSL
ARB_shadow		R25+	X	X	X	1.4 functionality
ARB_texture_border_clamp		X	X	X	X	1.3 functionality
ARB_texture_compression	X	X	X	X	X	1.3 functionality
ARB_texture_cube_map	X	X	X	X	X	1.3 functionality
ARB_texture_env_add	X	X	X	X	X	1.3 functionality
ARB_texture_env_combine	X	X	X	X	X	1.3 functionality
ARB_texture_env_crossbar						see explanation
ARB_texture_env_dot3	X	X	X	X	X	1.3 functionality
ARB_texture_mirrored_repeat	R40	R40	X	X	X	1.4, same as IBM
ARB_texture_non_power_of_two				X	X	2.0 functionality
ARB_texture_rectangle	R62	R60+	R62	R62	X	
ARB_transpose_matrix	X	X	X	X	X	1.3 functionality
ARB_vertex_buffer_object	R65	R65	R65	R65	X	1.5 functionality
ARB_vertex_program	R40+	R40+	X	X	X	
ARB_vertex_shader	R60	R60	R60	R60	X	2.0 functionality, GLSL
ARB_window_pos	R40	R40	X	X	X	1.4 functionality
ATI_draw_buffers				X	X	
ATI_texture_float				X	X	
ATI_texture_mirror_once				X	X	use EXT_texture_mirror_clamp
EXT_abgr	X	X	X	X	X	
EXT_bgra	X	X	X	X	X	1.2 functionality
EXT_bindable_uniform					X	GLSL extension
EXT_blend_color	X	X	X	X	X	1.4 functionality
EXT_blend_equation_separate				R60	X	2.0 functionality
EXT_blend_func_separate			X	X	X	1.4 functionality
EXT_blend_minmax	X	X	X	X	X	1.4 functionality
EXT_blend_subtract	X	X	X	X	X	1.4 functionality
EXT_Cg_shader	R60	R60	R60	R60	X	Cg through GLSL API
EXT_clip_volume_hint	R20+					
EXT_compiled_vertex_array	X	X	X	X	X	
EXT_depth_bounds_test			R50	X	X	NV35, NV36, NV4x in hw only
EXT_draw_buffers2					X	ARB_draw_buffers extension
EXT_draw_instanced					X	
EXT_draw_range_elements	R20	R20	X	X	X	1.2 functionality
EXT_fog_coord	X	X	X	X	X	1.4 functionality
EXT_framebuffer_blit			R95	R95	X	
EXT_framebuffer_multisample			R95	R95	X	
EXT_framebuffer_object			R75	R75	X	
EXT_framebuffer_sRGB					X	
EXT_geometry_shader4					X	GLSL extension
EXT_gpu_program_parameters	R95	R95	R95	R95	X	
EXT_gpu_shader4					X	GLSL extension
EXT_multi_draw_arrays	R25	R25	X	X	X	1.4 functionality
EXT_packed_depth_stencil			R80	X	X	
EXT_packed_float					X	
EXT_packed_pixels	X	X	X	X	X	1.2 functionality

Extension	NV1*x*	NV2*x*	*NV3x*	*NV4x*	*G8x*	Notes
EXT_paletted_texture	X	X	X			no NV4x hw support
EXT_pixel_buffer_object	R55	R55	R55	X	X	2.1 functionality
EXT_point_parameters	X	X	X	X	X	1.4 functionality
EXT_rescale_normal	X	X	X	X	X	1.2 functionality
EXT_secondary_color	X	X	X	X	X	1.4 functionality
EXT_separate_specular_color	X	X	X	X	X	1.2 functionality
EXT_shadow_funcs		R25+	X	X	X	1.5 functionality
EXT_shared_texture_palette	X	X	X			no NV4x hw support
EXT_stencil_clear_tag				R70		NV44 only
EXT_stencil_two_side			X	X	X	2.0 functionality
EXT_stencil_wrap	X	X	X	X	X	1.4 functionality
EXT_texture3D	sw	X	X	X	X	1.2 functionality
EXT_texture_array					X	
EXT_texture_buffer_object					X	
EXT_texture_compression_latc					X	
EXT_texture_compression_rgtc					X	
EXT_texture_compression_s3tc	X	X	X	X	X	
EXT_texture_cube_map	X	X	X	X	X	1.2 functionality
EXT_texture_edge_clamp	X	X	X	X	X	1.2 functionality
EXT_texture_env_add	X	X	X	X	X	1.3 functionality
EXT_texture_env_combine	X	X	X	X	X	1.3 functionality
EXT_texture_env_dot3	X	X	X	X	X	1.3 functionality
EXT_texture_filter_anisotropic	X	X	X	X	X	
EXT_texture_integer					X	
EXT_texture_lod	X	X	X	X	X	1.2 functionality; no spec
EXT_texture_lod_bias	X	X	X	X	X	1.4 functionality
EXT_texture_mirror_clamp				X	X	
EXT_texture_object	X	X	X	X	X	1.1 functionality
EXT_texture_shared_exponent					X	
EXT_texture_sRGB				X	X	2.1 functionality
EXT_timer_query		R80	R80	R80	X	
EXT_vertex_array	X	X	X	X	X	1.1 functionality
EXT_vertex_weighting	X	X				Discontinued
KTX_buffer_region	X	X	X	X	X	
HP_occlusion_test		R25	X	X	X	
IBM_rasterpos_clip	R40+	R40+	R40+	X	X	
IBM_texture_mirrored_repeat	X	X	X	X	X	1.4 functionality
KTX_buffer_region	X	X	X	X	X	use ARB_buffer_region
NV_blend_square	X	X	X	X	X	1.4 functionality
NV_conditional_render					X	
NV_copy_depth_to_color		R20	X	X	X	
NV_depth_buffer_float					X	
NV_depth_clamp		R25+	X	X	X	
NV_evaluators	R10	X				Discontinued
NV_fence	X	X	X	X	X	
NV_float_buffer			X	X	X	
NV_fog_distance	X	X	X	X	X	
NV_fragment_program			X	X	X	
NV_fragment_program_option			R55	X	X	NV_fp features for ARB_fp
NV_fragment_program2				X	X	
NV_fragment_program4					X	See NV_gpu_program4
NV_framebuffer_multisample_coverage			Nf	Nf	X	FBO extension
NV_geometry_program4					X	See NV_gpu_program4
NV_geometry_shader4					X	
NV_gpu_program4					X	
NV_half_float			X	X	X	
NV_light_max_exponent	X	X	X	X	X	
NV_multisample_filter_hint		X	X	X	X	
NV_occlusion_query		R25	X	X	X	
NV_packed_depth_stencil	R10+	R10+	X	X	X	
NV_parameter_buffer_object					X	See NV_gpu_program4
NV_pixel_data_range	R40	R40	X	X	X	
NV_point_sprite	R35+	R25	X	X	X	
NV_present_video					R165	SDI Quadro only

Extension	NV1*x*	NV2*x*	*NV3x*	*NV4x*	*G8x*	Notes
NV_primitive_restart			X	X	X	
NV_register_combiners	X	X	X	X	X	
NV_register_combiners2		X	X	X	X	
NV_texgen_emboss	X					Discontinued
NV_texgen_reflection	X	X	X	X	X	use 1.3 functionality
NV_texture_compression_vtc		X	X	X	X	
NV_texture_env_combine4	X	X	X	X	X	
NV_texture_expand_normal			X	X	X	
NV_texture_rectangle	X	X	X	X	X	
NV_texture_shader		X	X	X	X	
NV_texture_shader2		X	X	X	X	
NV_texture_shader3		R25	X	X	X	only NV25 and up in HW
NV_transform_feedback					X	
NV_vertex_array_range	X	X	X	X	X	
NV_vertex_array_range2	R10	R10	X	X	X	
NV_vertex_program	R10	X	X	X	X	
NV_vertex_program1_1	R25	R25	X	X	X	
NV_vertex_program2			X	X	X	
NV_vertex_program2_option			R55	X	X	
NV_vertex_program3				X	X	
NV_vertex_program4					X	See NV_gpu_program4
S3_s3tc	X	X	X	X	X	no spec; use EXT_t_c_s3tc
SGIS_generate_mipmap	R10	X	X	X	X	1.4 functionality
SGIS_multitexture	X	X				use 1.3 version
SGIS_texture_lod	X	X	X	X	X	1.2 functionality
SGIX_depth_texture		X	X	X	X	use 1.4 version
SGIX_shadow		X	X	X	X	use 1.4 version
SUN_slice_accum	R50	R50	R50	X	X	accelerated on NV3x/NV4x
GLX_EXT_texture_from_pixmap				X	X	GLX
GLX_NV_swap_group			X	X	X	GLX, framelock Quadro only
GLX_NV_video_out			X	X	X	GLX, SDI Quadro only
WGL_ARB_buffer_region	X	X	X	X	X	Win32
WGL_ARB_extensions_string	X	X	X	X	X	Win32
WGL_ARB_make_current_read	R55	R55	R55	X	X	
WGL_ARB_multisample		X	X	X	X	see ARB_multisample
WGL_ARB_pixel_format	R10	X	X	X	X	Win32
WGL_ARB_pbuffer	R10	X	X	X	X	Win32
WGL_ARB_render_texture	R25	R25	X	X	X	Win32
WGL_ATI_pixel_format_float				X	X	Win32
WGL_EXT_extensions_string	X	X	X	X	X	Win32
WGL_EXT_swap_control	X	X	X	X	X	Win32
WGL_NV_float_buffer			X	X	X	Win32, see NV_float_buffer
WGL_NV_gpu_affinity				R95	X	Win32 SLI
WGL_NV_render_depth_texture		R25	X	X	X	Win32
WGL_NV_render_texture_rectangle	R25	R25	X	X	X	Win32
WGL_NV_swap_group			X	X	X	Win32, framelock Quadro only
WGL_NV_video_out			X	X	X	Win32, SDI Quadro only
WIN_swap_hint	X	X	X	X	X	Win32, no spec

Key for table entries:

X = *supported*

Q = *requires particularly Quadro cards*

sw = *supported by software rasterization (expect poor performance)*

Nf = *Extension advertised but rendering functionality not available*

R10 = *introduced in the Release 10 OpenGL driver (not supported by earlier drivers)*

R20 = *introduced in the Detanator XP (also known as Release 20) OpenGL driver (not supported by earlier drivers)*

R20+ = *introduced after the Detanator XP (also known as Release 20) OpenGL driver (not supported by earlier drivers)*

R25 = *introduced in the GeForce4 launch (also known as Release 25) OpenGL driver (not supported by earlier drivers)*

R25+ = *introduced after the GeForce4 launch (also known as Release 25) OpenGL driver (not supported by earlier drivers)*

R35 = *post-GeForce4 launch OpenGL driver release (not supported by earlier drivers)*

R40 = Detonator 40 release, August 2002.

R40+ = *introduced after the Detanator 40 (also known as Release 40) OpenGL driver (not supported by earlier drivers)*

R50 = Detonator 50 release

R55 = Detonator 55 release

R60 = Detonator 60 release, May 2004

R65 = Release 65

R70 = Release 70

R80 = Release 80

R95 = Release 95

no spec = *no suitable specification available*

Discontinued = earlier drivers (noted by 25% gray entries) supported this extension but support for the extension is discontinued in current and future drivers

Notices:

Emulation: While disabled by default, older GPUs can support extensions supported in hardware by newer GPUs through a process called emulation though any functionality unsupported by the older GPU must be emulated via software. For more details see: http://developer.nvidia.com/object/nvemulate.html

Warning: The extension support columns are based on the latest & greatest NVIDIA driver release (unless otherwise noted). Check your GL_EXTENSIONS string with glGetString at run-time to determine the specific supported extensions for a particular driver version.

Discontinuation of support: NVIDIA drivers from release 95 no longer support NV1x- and NV2x-based GPUs.

Name

    ARB_color_buffer_float

Name Strings

    GL_ARB_color_buffer_float
    WGL_ARB_pixel_format_float
    GLX_ARB_fbconfig_float

Contributors

    Pat Brown
    Jon Leech
    Rob Mace
    V Moya
    Brian Paul

Contact

    Dale Kirkland, NVIDIA (dkirkland 'at' nvidia.com)

Status

    Complete. Appprove by the ARB on October 22, 2004.

Version

    Based on the ATI_pixel_format_float extension, verion 5
    Enables based on work by Pat Brown from the color_clamp_control proposal

    Last Modified Date: February 7, 2006
    Version   6

Number

    ARB Extension #39

Dependencies

    This extension is written against the OpenGL 2.0 Specification
    but will work with the OpenGL 1.5 Specification.

    WGL_ARB_pixel_format is required.

    This extension interacts with ARB_fragment_program.

    This extension interacts with ARB_fragment_shader.

    This extension interacts with NV_float_buffer.

    This extension interacts with ATI_pixel_format_float.

Overview

    The standard OpenGL pipeline is based on a fixed-point pipeline.
    While color components are nominally floating-point values in the

pipeline, components are frequently clamped to the range [0,1] to accomodate the fixed-point color buffer representation and allow for fixed-point computational hardware.

This extension adds pixel formats or visuals with floating-point RGBA color components and controls for clamping of color components within the pipeline.

For a floating-point RGBA pixel format, the size of each float components is specified using the same attributes that are used for defining the size of fixed-point components. 32-bit floating-point components are in the standard IEEE float format. 16-bit floating-point components have 1 sign bit, 5 exponent bits, and 10 mantissa bits.

Clamping control provides a way to disable certain color clamps and allow programs, and the fixed-function pipeline, to deal in unclamped colors. There are controls to modify clamping of vertex colors, clamping of fragment colors throughout the pipeline, and for pixel return data.

The default state for fragment clamping is "FIXED_ONLY", which has the behavior of clamping colors for fixed-point color buffers and not clamping colors for floating-pont color buffers.

Vertex colors are clamped by default.

IP Status

SGI owns US Patent #6,650,327, issued November 18, 2003. SGI believes this patent contains necessary IP for graphics systems implementing floating point (FP) rasterization and FP framebuffer capabilities.

SGI will not grant the ARB royalty-free use of this IP for use in OpenGL, but will discuss licensing on RAND terms, on an individual basis with companies wishing to use this IP in the context of conformant OpenGL implementations. SGI does not plan to make any special exemption for open source implementations.

Contact Doug Crisman at SGI Legal for the complete IP disclosure.

Issues

1. *How is this extension different from the ATI_pixel_format_float extension?*

   RESOLVED: By default, this extension behaves like the ATI_pixel_format_float, but also adds additional controls for color clamping.

2. *Should the clamp controls be automatically inferred based on the format of the color buffer or textures used?*

   RESOLVED: Explicit controls should be supported -- this allows the use of floating-point buffers to emulate fixed-point operation, and allows for operating on unclamped values even when rendering to a fixed-point framebuffer.

However, a default clamping mode called "FIXED_ONLY" is defined that enables clamping only when rendering to a fixed-point color buffer, which is the default for fragment processing. This is done to maintain compatibility with previous extensions (ATI_pixel_format_float), and to allow applications to switch between fixed- and floating-point color buffers without having to change the clamping mode on each switch.

3. *How does the clamping control affect the blending equation?*

   RESOLVED: For fixed-point color buffers, the inputs and the result of the blending equation are clamped. For floating-point color buffers, no clamping occurs.

4. *Should the requirements for the representable range of color components be increased?*

   RESOLVED: No. Such a spec change would be complicated, since the required precision may vary based on color buffer precision. Despite the fact that there is no spec requirement, GL implementations should have at least as much precision/range in their colors as can be found in the framebuffer.

5. *Should the vertex color clamping control apply to RasterPos? WindowPos?*

   RESOLVED: Yes to both. RasterPos is processed just like a vertex, so the vertex color clamping control applies automatically. The WindowPos language in the OpenGL 2.0 specification explicitly refers to color clamping. Instead, we modify the language to perform normal processing, but with lighting forced off. This will result in the color clamping logic applying.

6. *What control should apply to DrawPixels RGBA components?*

   RESOLVED: The fragment color clamp control.

7. *Should this extension modify the clamping of the texture environment color components? TEXTURE_ENV_COLOR components are currently specified to be clamped to [0,1] when TexEnv is called.*

   RESOLVED: Yes. The texture environment color is no longer clamped when specified. If fragment color clamping is enabled, it will be clamped to [0,1] on use.

8. *In texture environment application, should color components used as an interpolation factor (e.g., alpha) be clamped to [0,1]?*

   RESOLVED: No. For interpolation-type blends, the weighting factor is normally in the range [0,1]. But the math is well-defined in the cases where it falls outside this range. When fragment color clamping is enabled, all sources are clamped to [0,1], so this is not an issue.

9. In the COMBINE texture environment mode, should any of the source argument operands be clamped to [0,1] even when fragment clamping is disabled? For example, ONE_MINUS_* mappings are simple in a fixed-point pipeline are simple, but more complicated in a floating-point one.

RESOLVED: No. The math behind ONE_MINUS_* is well-defined for all inputs.

10. Should the clamping controls affect the texture comparison mode for shadow mapping?

RESOLVED: No. The r coordinate should still be clamped to [0,1] to match the depth texture. The result of the comparison will naturally lie in the range [0,1].

11. Should the clamping controls affect the result of color sum?

RESOLVED: Yes.

12. Should the clamping controls affect the computed fog factor?

RESOLVED: No. The fog factor is not a color -- it is used to blend between the fragment color and the fog color. The factor should always be clamped to [0,1].

13. Should this extension modify the clamping of the fog color components? FOG_COLOR components are specified to be clamped to [0,1] when Fogfv is called.

RESOLVED: Yes. Fog color components are no longer clamped when specified, but will be clamped when fog is applied if fragment color clamping is enabled.

14. How does this extension interact with antialiasing application (Section 3.12 of the OpenGL 2.0 spec)?

RESOLVED: Multiply floating-point alpha by coverage, even if the alpha value is not being used as opacity. If applications don't want this multiplication, they should not render antialiased primitives. No spec language changes are needed here.

15. How does this extension interact with multisample point fade (Section 3.13 of the OpenGL 2.0 spec)?

RESOLVED: Multiply floating-point alpha by the fade factor, even if the alpha value is not being used as opacity. If applications don't want this multiplication, they should not use multisample point fade. No spec language changes are needed here.

16. *Should this extension modify the clamping of the alpha test reference value?*

    RESOLVED: Yes. The reference value is not clamped when specified, by may be clamped when it is used.

17. *Should this extension modify the clamping of the constant blend color components?*

    RESOLVED: Yes. The blend color is not clamped when specified. When rendering to a fixed-point framebuffer, the blend color will be clamped as part of the blending operation.

18. *Should this extension modify the clamping of clear colors?*

    RESOLVED: Yes. The clear color is not clamped when specified. When clearing color buffers, the clear color is converted to the format of the color buffer.

19. *Should we provide a control to disable implicit clamping of ReadPixels data? If so, how should it be specified?*

    RESOLVED: Yes. It is explicitely controlled by the target CLAMP_READ_COLOR_ARB of the ClampColorARB function and clamps the color during the final conversion.

20. *How does this extension interact with CopyPixels?*

    RESOLVED: It has no special interaction. CopyPixels is specified as roughly a ReadPixels/DrawPixels sequence, but the read color clamp modified by this specification occur during final conversion and therefore would not apply. The fragment color clamp does affect the DrawPixels portion of the operation, however. The net result is that calling CopyPixels with a floating-point framebuffer will clamp color components if fragment color clamping is enabled.

21. *Should these clamping controls interact with PushAttrib and PopAttrib? If so, what group should they belong to?*

    RESOLVED: For consistency, yes. Historically, all enables are pushed and popped with both the enable bit and a second bit corresponding to the function performed by the enable. The present spec calls for pushing the vertex color clamp with the lighting group and the fragment and read color clamp with the color-buffer group (for lack of a better choice).

22. *Should this extension require a floating-point color buffer or texture?*

    RESOLVED: No. This extension provides the ability to pass an unclamped color between vertex and fragment programs/shaders, which may be useful. This was possible prior to this extension, by passing the color data as texture coordinates or named varying variables (for vertex/fragment shaders).

23. *Does this extension interact with the ARB_vertex_program or ARB_vertex_shader extensions?*

    RESOLVED: Only in the most trivial way. Both of these extensions refer to the color clamping logic (Section 2.14.6 in the OpenGL 2.0 specification). This extension modifies that logic to be under control of the CLAMP_VERTEX_COLOR_ARB enable. It follows that this enable also controls the clamping of vertex program or vertex shader results.

24. *Does this extension interact with the ARB_fragment_program or ARB_fragment_shader extensions?*

    RESOLVED: Yes. The only interaction is that the fragment color clamp enable determines if the final color(s) produced by the fragment program/shader has its components clamped to [0,1].

    However, the fragment color clamp enable affects only the final result; it does NOT affect any computations performed during program execution. Note that the same clamping can be done explicitly in a fragment program or shader. ARB_fragment_program provides the "_SAT" opcode suffix to clamp instruction results to [0,1].

25. *Should this extension modify the clamping of the texture border color components?*

    RESOLVED: Not by this extension. See the ARB_texture_float extension.

26. *When using vertex and fragment programs/shaders, should color clamping be specified in the shader instead?*

    RESOLVED: No. All the existing program/shader extensions call for the color outputs to be clamped to [0,1], except that previous floating-point color buffer extensions disabled the clamp of fragment program/shader outputs.

    While it would be straightforward to have required that vertex or fragment programs manually clamp their outputs if desired, adding such a requirement at this point would pose compatibility issues. It would probably require introduction of a special directive to indicate that colors are unclamped.

    If a GL implementation internally performs color clamping in a vertex or fragment program, it may be necessary to recompile the program if the corresponding clamp enable changes.

27. *If certain colors in the OpenGL state vector were clamped in previous versions of the spec, but now have the clamping removed, do queries need to return clamped values for compatibility with older GL versions? Should we add new query tokens to return unclamped values?*

    RESOLVED: To minimize impact on this specification while allowing for compatibility with older GL versions, the values of the vertex/fragment color clamp enables should affect queries of such state. If the corresponding color clamp is enabled, components will be clamped to [0,1] when returned. Since color clamping is enabled by default for fixed-point color buffers, the removal of the clamps will not be observable by applications unless they disable one or both clamps or choose a floating-point buffer (which will not happen for "old" applications).

Note that this spec relaxes the clamp on the current raster color, but we don't need to add a clamp on the corresponding query. The current raster color is clamped when the GL computes it, unless vertex color clamping is disabled by the application.

28. *At what precision should alpha test be carried out? At the precision of the framebuffer? Or some other unspecified precision? What happens if you have a framebuffer with no alpha?*

RESOLVED: No specific precision requirements are added, except that the reference value used in the alpha test should be converted to the same precision and in the same manner as the fragment's alpha. This requirement is intended to avoid cases where the act of converting the alpha value of a fragment to fixed-point (or lower-precision floating-point) might change the result of the test.

29. *How does this extension interact with accumulation buffers?*

RESOLVED: This extension does not modify the nature of accumulation buffers. Adding semantics for floating-point accumulation buffers is left for a possible future extension. The clamp on the RETURN operation is controlled by the fragment color clamp enable.

30. *How does this extension interact with OpenGL FEEDBACK mode?*

RESOLVED: OpenGL FEEDBACK mode returns colors after clipping, which is done after the vertex color clamping. Therefore, the colors returned will be clamped to [0,1] if and only if vertex color clamping is enabled. No spec language changes are necessary.

31. *Should we relax the language in Section 2.14.9 (Final Color Processing) to not require conversion to fixed-point?*

RESOLVED: Adding floating-point vertex colors requires that this language be modified. Even for the clamped case, it seems reasonable for implementations to simply clamp a floating-point value to [0,1] without converting to a fixed-point representation. This specification makes converting colors to fixed-point optional. Colors will obviously still be converted to fixed-point eventually if the framebuffer is fixed-point.

32. *What should be done about the "preserving the bits" requirement for Color*{ub,us,ui} commands in Section 2.14.9?*

RESOLVED: If colors are represented as floats internally and the frame-buffer is fixed-point, do we require that the MSBs of fixed-point colors that don't go through lighting, and non-trivial interpolation, or any non-trivial fragment operations show up in the MSBs of the framebuffer?

33. *How does this extension interact with multisample ALPHA_TO_COVERAGE, where an alpha value expected to be in the range [0,1] is turned into a set of coverage bits?*

    UNRESOLVED: For the purposes of generating sample coverage from fragment alpha, the alpha values are effectively clamped to [0,1]. Negative alpha values correspond to no coverage; alpha values greater than one correspond to full coverage.

34. *What happens if there are no color buffers in the framebuffer and a clamp control is set to FIXED_ONLY?*

    RESOLVED: The present language treats a zero-bit color buffer as fixed-point.

35. *Should the clamping of fragment shader output gl_FragData[n] be controlled by the fragment color clamp.*

    RESOLVED: Since the destination of the FragData is a color buffer, the fragment color clamp control should apply.

36. *Should logical operations be disabled for floating-point color buffers.*

    RESOLVED: Yes. This matches the behavior in the ATI specification.

37. *Is it expected that a floating-point color read from a floating-point color buffer exactly match a floating-point color in a fragment? Will the alpha test of GL_EQUAL be expected to work?*

    RESOLVED: This behavior is not required by this extension. Floating-point data may have different precision at different parts of the pipeline.

38. *How does this extension handle the case where a floating-point and a fixed-point buffer exists?*

    RESOLVED: For vertex colors, clamping occurs if any color buffer are floating point. Fragment colors are handled based on the format (fixed or float) of the color buffer that they will be drawn to.

New Procedures and Functions

```
    void ClampColorARB(enum target, enum clamp);
```

New Tokens

```
    Accepted by the <pname> parameters of GetBooleanv, GetIntegerv,
    GetFloatv, and GetDoublev:

        RGBA_FLOAT_MODE_ARB                         0x8820

    Accepted by the <target> parameter of ClampColorARB and the <pname>
    parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev.

        CLAMP_VERTEX_COLOR_ARB                      0x891A
        CLAMP_FRAGMENT_COLOR_ARB                    0x891B
        CLAMP_READ_COLOR_ARB                        0x891C

    Accepted by the <clamp> parameter of ClampColorARB.

        FIXED_ONLY_ARB                              0x891D
        FALSE
        TRUE

    Accepted as a value in the <piAttribIList> and <pfAttribFList>
    parameter arrays of wglChoosePixelFormatARB, and returned in the
    <piValues> parameter array of wglGetPixelFormatAttribivARB, and the
    <pfValues> parameter array of wglGetPixelFormatAttribfvARB:

        WGL_TYPE_RGBA_FLOAT_ARB                     0x21A0

    Accepted as values of the <render_type> arguments in the
    glXCreateNewContext and glXCreateContext functions

        GLX_RGBA_FLOAT_TYPE                         0x20B9

    Accepted as a bit set in the GLX_RENDER_TYPE variable

        GLX_RGBA_FLOAT_BIT                          0x00000004
```

Additions to Chapter 2 of the OpenGL 2.0 Specification (OpenGL Operation)

```
    Add a new Section 2.1.2, (p. 6):
```

2.1.2 16-Bit Floating-Point Numbers

```
    A 16-bit floating-point number has a 1-bit sign (S), a 5-bit
    exponent (E), and a 10-bit mantissa (M).  The value of a 16-bit
    floating-point number is determined by the following:

        (-1)^S * 0.0,                        if E == 0 and M == 0,
        (-1)^S * 2^-14 * (M / 2^10),         if E == 0 and M != 0,
        (-1)^S * 2^(E-15) * (1 + M/2^10),    if 0 < E < 31,
        (-1)^S * INF,                        if E == 31 and M == 0, or
        NaN,                                 if E == 31 and M != 0,

    where

        S = floor((N mod 65536) / 32768),
        E = floor((N mod 32768) / 1024), and
        M = N mod 1024.
```

Implementations are also allowed to use any of the following alternative encodings:

```
    (-1)^S * 0.0,                        if E == 0 and M != 0,
    (-1)^S * 2^(E-15) * (1 + M/2^10),    if E == 31 and M == 0, or
    (-1)^S * 2^(E-15) * (1 + M/2^10),    if E == 31 and M != 0,
```

Any representable 16-bit floating-point value is legal as input to a GL command that accepts 16-bit floating-point data. The result of providing a value that is not a floating-point number (such as infinity or NaN) to such a command is unspecified, but must not lead to GL interruption or termination. Providing a denormalized number or negative zero to GL must yield predictable results.

Modify Section 2.13 (Current Raster Position), p. 54

(modify last paragraph on p. 55) Lighting, texture coordinate generation and transformation, and clipping are not performed by the WindowPos functions. Instead, in RGBA mode, the current raster color and secondary color are obtained from the current color and secondary color, respectively. If vertex color clamping is enable, the current raster color and secondary color are clamped to [0, 1]. In color index mode, the current raster color index is set to the current color index. The current raster texture coordinates are set to the current texture coordinates, and the valid bit is set.

Modify Section 2.14 (Colors and Coloring), p. 57

(modify last paragraph on p.57) ... After lighting, RGBA colors are optionally clamped to the range [0,1]. ...

Modify Section 2.14.6 (Clamping or Masking), p. 69

(modify first and second paragraphs of section) When the GL is in RGBA mode and vertex color clamping is enabled, all components of both primary and secondary colors are clamped to the range [0,1] after lighting. If color clamping is disabled, the primary and secondary colors are unmodified. Vertex color clamping is controlled by calling

```
    void ClampColorARB(enum target, enum clamp)
```

with a <target> set to CLAMP_VERTEX_COLOR_ARB. If <clamp> is TRUE, vertex color clamping is enabled; if <clamp> is FALSE, vertex color clamping is disabled. If <clamp> is FIXED_ONLY_ARB, vertex color clamping is enabled if all enabled color buffers have fixed-point components.

For a color index, the index is first converted to...

(add paragraph at the end of the section) The state required for color clamping is an enumerant. Vertex color clamping is initially TRUE.

Replace Section 2.14.9 (Final Color Processing), p. 71

In RGBA mode with vertex color clamping disabled, the floating-point RGBA components are not modified.

In RGBA mode with vertex clamping enabled, each color component (already clamped to [0,1]) may be converted (by rounding to nearest) to a fixed-point value with m bits. We assume that the fixed-point representation used represents each value k/(2^m - 1), with k in the set {0, 1, . . . , 2^m - 1}, as k (e.g. 1.0 is represented in binary as a string of all ones). m must be at least as large as the number of bits in the corresponding component of the framebuffer. m must be at least 2 for A if the framebuffer does not contain an A component, or if there is only 1 bit of A in the framebuffer. GL implementations are not required to convert clamped color components to fixed-point.

Because a number of the form k/(2^m - 1) may not be represented exactly as a limited-precision floating-point quantity, we place a further requirement on the fixed-point conversion of RGBA components. Suppose that lighting is disabled, the color associated with a vertex has not been clipped, and one of Colorub, Colorus, or Colorui was used to specify that color. When these conditions are satisfied, an RGBA component must convert to a value that matches the component as specified in the Color command: if m is less than the number of bits b with which the component was specified, then the converted value must equal the most significant m bits of the specified value; otherwise, the most significant b bits of the converted value must equal the specified value.

In color index mode, a color index is converted (by rounding to nearest) to a fixed-point value with at least as many bits as there are in the color index portion of the framebuffer.

Additions to Chapter 3 of the OpenGL 2.0 Specification (Rasterization)

Modify Section 3.6.4 (Rasterization of Pixel Rectangles), p. 126

(modify next-to-last paragraph, p.136, "Final Conversion") ... For RGBA components, if fragment color clamping is enabled, each element is clamped to [0,1], and may be converted to fixed-point according to the rules given in section 2.14.9 (Final Color Processing). If fragment color clamping is disabled, RGBA components are unmodified. Fragment color clamping is controlled using ClampColorARB, as described in section 2.14.6, with a <target> of CLAMP_FRAGMENT_COLOR_ARB.

(add new paragraph at the end of "Final Conversion", p.137) The state required for fragment color clamping is an enumerant. Fragment color clamping is initially set to FIXED_ONLY_ARB.

Modify Section 3.8.13 (Texture Environments and Functions), p.182

(modify third paragraph, p. 183, removing clamping language) ...TEXTURE_ENV_COLOR is set to an RGBA color by providing four single-precision floating-point values. If integers are provided for TEXTURE ENV COLOR, then they are converted to floating-point as specified in table 2.9 for signed integers.

(replace the sixth paragraph of p. 183) If fragment color clamping is enabled, all of these color values, including the results, are clamped to the range [0,1]. If fragment color clamping is disabled, the values are not clamped. The texture functions are specified in tables 3.22, 3.23, and 3.24.

(modify seventh paragraph of p. 183) ... ALPHA_SCALE, respectively. If fragment color clamping is enabled, the arguments and results used in table 3.24 are clamped to [0,1]. Otherwise, the results are unmodified.

Modify Section 3.9 (Color Sum), p. 191

(modify second paragraph) ... the A component of c_sec is unused. If color sum is disabled, then c_pri is assigned to c. The components of c are then clamped to the range [0,1] if and only if fragment color clamping is enabled.

Modify Section 3.10 (Fog), p. 191

(modify fourth paragraph, p. 192, removing clamping language) ...If these are not floating-point values, then they are converted to floating-point using the conversion given in table 2.9 for signed integers. If fragment color clamping is enabled, the components of C_r and C_f and the result C are clamped to the range [0,1] before the fog blend is performed.

Modify Section 3.11.2 (Shader Execution), p. 194

(modify Shader Inputs, first paragraph, p. 196) The built-in variables gl_Color and gl_SecondaryColor hold the R, G, B, and A components, respectively, of the fragment color and secondary color. If the primary color or the secondary color components are represented by the GL as fixed-point values, they undergo an implied conversion to floating-point. This conversion must leave the values 0 and 1 invariant. Floating-point color components (resulting from a disabled vertex color clamp) are unmodified.

(modify Shader Outputs, first paragraph, p. 196) ... These are gl_FragColor, gl_FragData[n], and gl_FragDepth. If fragment clamping is enabled, the final fragment color values or the final fragment data values written by a fragment shader are clamped to the range [0, 1] and then may be converted to fixed-point as described in section 2.14.9. If fragment clamping is disabled, the final fragment color values or the final fragment data values are not modified. The final fragment depth...

Additions to Chapter 4 of the OpenGL 2.0 Specification (Per-Fragment Operations and the Framebuffer)

Modify Chapter 4 Introduction, (p. 198)

(modify third paragraph, p. 198) Color buffers consist of either unsigned integer color indices, R, G, B and optionally A unsigned integer values, or R, G, B, and optionally A floating-point values. The number of bitplanes...

Modify Section 4.1.3 (Multisample Fragment Operations), p. 200

(modify last paragraph, p. 200) ...and all 0's corresponding to all alpha values being 0. The alpha values used to generate a coverage value are clamped to the range [0,1]. It is also intended ...

Modify Section 4.1.5 (Alpha Test), p. 201

(modify first paragraph of section, deleting clamping of reference value) ... The test is controlled with

```
   void AlphaFunc(enum func, float ref);
```

func is a symbolic constant indicating the alpha test function; ref is a reference value. When performing the alpha test, the GL will convert the reference value to the same representation as the the fragment's alpha value (floating-point or fixed-point). For fixed-point, the reference value is converted according to the rules given for an A component in section 2.14.9 and the fragment's alpha value is rounded to the nearest integer. The possible ...

Modify Section 4.1.8 (Blending), p. 205

(modify first paragraph, p. 206) Source and destination values are combined according to the blend equation, quadruplets of source and destination weighting factors determined by the blend functions, and a constant blend color to obtain a new set of R, G, B, and A values, as described below.

If the color buffer is fixed-point, the components of the source and destination values and blend factors are clamped to [0, 1] prior to evaluating the blend equation, the components of the blending result are clamped to [0,1] and converted to fixed-point values in the manner described in section 2.14.9. If the color buffer is floating-point, no clamping occurs. The resulting four values are sent to the next operation.

(modify fifth paragraph, p. 206) Fixed-point destination (framebuffer) components are taken to be fixed-point values represented according to the scheme given in section 2.14.9 (Final Color Processing). Constant color components, floating-point destination components, and source (fragment) components are taken to be floating point values. If source components are represented internally by the GL as either fixed-point values they are also interepreted according to section 2.14.9.

(modify Blend Color section removing the clamp, p. 209) The constant color C_c to be used in blending is specified with the command

```
   void BlendColor(float red, float green, float blue, float alpha);
```

The constant color can be used in both the source and destination blending functions.

Replace Section 4.1.9 (Dithering), p. 209

Dithering selects between two representable color values or indices. A representable value is a value that has an exact representation in the color buffer. In RGBA mode dithering selects, for each color component, either the most positive representable color value (for that particular color component) that is less than or equal to the incoming color component value, c, or the most negative representable color value that is greater than or equal to c. The selection may depend on the x_w and y_w coordinates of the pixel, as well as on the exact value of c. If one of the two values does not exist, then the selection defaults to the other value.

In color index mode dithering selects either the largest representable index that is less than or equal to the incoming color value, c, or the smallest representable index that is greater than or equal to c. If one of the two indices does not exist, then the selection defaults to the other value.

Many dithering selection algorithms are possible, but an individual selection must depend only on the incoming color index or component value and the fragment's x and y window coordinates. If dithering is disabled, then each incoming color component c is replaced with the most positive representable color value (for that particular component) that is less than or equal to c, or by the most negative representable value, if no representable value is less than or equal to c; a color index is rounded to the nearest representable index value.

Dithering is enabled with Enable and disabled with Disable using the symbolic constant DITHER. The state required is thus a single bit. Initially dithering is enabled.

Section 4.1.10 (Logical Operation), p. 210

(insert after the first sentence, p. 210) Logical operation has no effect on a floating-point destination color buffer. However, if COLOR_LOGIC_OP is enabled, blending is still disabled.

Modify Section 4.2.3 (Clearing the Buffers), p. 215

(modify second paragraph, p. 216, removing clamp of clear color)

```
   void ClearColor(float r, float g, float b, float a);
```

sets the clear value for the color buffers in RGBA mode.

(add to the end of first partial paragraph, p. 217) ... then a Clear directed at that buffer has no effect. Fixed-point RGBA color buffers are cleared to a color values derived by taking the clear color, clamping to [0,1], and converting to fixed-point according to the rules of section 2.14.9.

Modify Section 4.2.4 (The Accumulation Buffer), p. 217

(modify second paragraph in section, p. 217) ... Using ACCUM obtains R, G, B, and A components from the color buffer currently selected for reading (section 4.3.2). If the color buffer is fixed-point, each component is considered as a fixed-point value in [0,1] (see section 2.14.9) and is converted to floating-point. Each result is then multiplied ...

(modify second paragraph on p. 218) The RETURN operation takes each color value from the accumulation buffer and multiplies each of the R, G, B, and A components by <value>. If fragment color clamping is enabled, the results are then clamped to the range [0,1]. ...

Modify Section 4.3.2 (Reading Pixels), p. 219

(modify paragraph at top of page, p. 222) ... For a fixed-point color buffer, each element is taken to be a fixed-point value in [0, 1] with m bits, where m is the number of bits in the corresponding color component of the selected buffer (see section 2.14.9). For floating-point color buffer, the elements are unmodified.

(modify second paragraph of "Final Conversion", p. 222) For an RGBA color, if <type> is not FLOAT, or if the CLAMP_READ_COLOR_ARB is TRUE, or CLAMP_READ_COLOR_ARB is FIXED_ONLY_ARB and the selected color buffer is a fixed-point buffer, each component is first clamped to [0,1]. Then the appropriate conversion...

Additions to Chapter 5 of the OpenGL 2.0 Specification (Special Functions)

None

Additions to Chapter 6 of the OpenGL 2.0 Specification (State and State Requests)

Modify Section 6.1.2, Data Conversions, p. 245

(add new paragraph at the end of the section, p. 245) If fragment color clamping is enabled, querying of the texture border color, texture environment color, fog color, alpha test reference value, blend color, and RGBA clear color will clamp the corresponding state values to [0,1] before returning them. This behavior provides compatibility with previous versions of the GL that clamped these values when specified.

Additions to Chapter 1 of the GLX 1.3 Specification (Overview)

None

Additions to Chapter 2 of the GLX 1.3 Specification (GLX Operation)

None

Additions to Chapter 3 of the GLX 1.3 Specification (Functions and Errors)

Replace Section 3.3.3 (p.12) Paragraph 4 to:

The attribute GLX_RENDER_TYPE has as its value a mask indicating what type of GLXContext a drawable created with the corresponding GLXFBConfig can be bound to. The following bit settings are supported: GLX_RGBA_BIT, GLX_RGBA_FOAT_BIT, GLX_COLOR_INDEX_BIT. If combinations of bits are set in the mask then drawables created with the GLXFBConfig can be bound to those corresponding types of rendering contexts.

Add to Section 3.3.3 (p.15) after first paragraph:

Note that floating point rendering is only supported for GLXPbuffer drawables. The GLX_DRAWABLE_TYPE attribute of the GLXFBConfig must have the GLX_PBUFFER_BIT bit set and the GLX_RENDER_TYPE attribute must have the GLX_RGBA_FLOAT_BIT set.

Modify Section 3.3.7 (p.25 Rendering Contexts) remove period at end of second paragraph and replace with:

; if render_type is set to GLX_RGBA_FLOAT_TYPE then a context that supports floating point RGBA rendering is created.

Additions to Chapter 4 of the GLX 1.3 Specification (Encoding on the X Byte Stream)

None

Additions to Chapter 5 of the GLX 1.3 Specification (Extending OpenGL)

None

Additions to Chapter 6 of the GLX 1.3 Specification (GLX Versions)

None

Additions to Chapter 7 of the GLX 1.3 Specification (Glossary)

None

Additions to the GLX Specification

Modify the bit field GLX_RENDER_TYPE to:

GLX_RENDER_TYPE
The type of pixel data. This bit field can have the following bit set: GLX_RGBA_BIT, GLX_RGBA_FLOAT_BIT, GLX_COLOR_INDEX_BIT

```
    Adds to the accepted values of the <render_type> argument
    in the glXCreateNewContext and glXCreateContextWithSGIX
    functions to:

    <render_type>
    Type of rendering context requested.  This argument
    can have the following values: GLX_RGBA_TYPE,
    GLX_RGBA_FLOAT_TYPE, GLX_COLOR_INDEX_TYPE
```

GLX Protocol

```
    The following rendering commands are sent to the server as part
    of a glXRender request:

    ClampColorARB
        2           12              rendering command length
        2           234             rendering command opcode
        4           CARD32          target
        4           CARD32          clamp
```

Additions to the WGL Specification

```
    Modify the values accepted by WGL_PIXEL_TYPE_ARB to:

        WGL_PIXEL_TYPE_ARB
        The type of pixel data. This can be set to WGL_TYPE_RGBA_ARB,
        WGL_TYPE_RGBA_FLOAT_ARB, or WGL_TYPE_COLORINDEX_ARB.
```

Dependencies on WGL_ARB_pixel_format

```
    The WGL_ARB_pixel_format extension must be used to determine a
    pixel format with float components.
```

Dependencies on ARB_fragment_program

```
    (modify 2nd paragraph of Section 3.11.4.4 language) If fragment
    color clamping is enabled, the fragment's color components are first
    clamped to the range [0,1] and are optionally converted to fixed
    point as in section 2.14.9.  If the fragment program does not write
    result.color, the color will be undefined in subsequent stages.
```

Dependencies on ARB_fragment_shader

```
    (modify 1st paragraph of Section 3.11.6 language) ... are
    gl_FragColor and gl_FragDepth.  If fragment color clamping is
    enabled, the final fragment color values written by a fragment
    shader are clamped to the range [0,1] and are optionally converted
    to fixed-point as described in section 2.14.9, Final Color
    Processing.  ...
```

Dependencies on NV_float_buffer

```
    Note that the WGL/GLX enumerants for the NV and ARB extensions
    do not have the same values, so it is possible to distinguish
    between "NV" and "ARB" pixel formats.
```

If NV_float_buffer and ARB_color_buffer_float are both supported, restrictions imposed by NV_float_buffer are removed. In particular, antialiasing application, multisample fragment operations, alpha test, and blending are all performed as specified in this extension. Additionally, it is not necessary to use a fragment program or shader to render to a floating-point color buffer allocated using the NV_float_buffer extension.

Note also that vertex color clamp portion of this extension does not interact with NV_float_buffer.

Dependencies on ATI_pixel_format_float

The basic policy of ATI_pixel_format_float regarding clamping is that vertex color clamping is unaffected (still enabled) and that fragment color clamping is automatically disabled when rendering to floating-point color buffers.

This extension is designed so that the defaults are compatible with the ATI_pixel_format_float, so there is no need for separate "ATI" and "ARB" floating-point pixel formats.

Errors

None

New State

(modify table 6.10, p. 271)

Get Value	Type	Get Command	Initial Value	Description	Sec.	Attribute
CLAMP_VERTEX_COLOR_ARB	B	GetIntegerv	TRUE	vertex color clamping	2.14.6	lighting/enable
CLAMP_FRAGMENT_COLOR_ARB	B	GetIntegerv	FIXED_ ONLY_ARB	fragment color clamping	2.14.6	color-buffer/enable
CLAMP_READ_COLOR_ARB	B	GetIntegerv	FIXED_ ONLY_ARB	read color clamping	2.14.6	color-buffer/enable

(modify table 6.33, p. 294)

Get Value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
RGBA_FLOAT_MODE_ARB	B	GetBooleanv	-	True if RGBA components are floats	2.7	-

New Implementation Dependent State

None

Revision History

Rev.	Date	Author	Changes
1	2/26/04	Kirkland	Initial version based on the ATI extension.

2	3/11/04	Kirkland	Changed spec to be both a GL and WGL spec. Updated language for float16 number handling. Added bit encodings for half values. Removed the clamped color query. Updated the language for dithering.
3	7/23/04	Kirkland	Added alternative encodings options for float16 format.
4	9/17/04	Kirkland	Merged the color clamp control spec with this spec. Updated to reference the OpenGL 2.0 spec. Added the specification for GLX.
5	10/1/04	Kirkland	Updated IP section. Reviewed by the ARB and closed all UNRESOLVED issues. Added an invariant that discusses the handling of the alpha test.
6	2/6/07	Jon Leech	Fix typos in enum naming.

Name

ARB_depth_texture

Name Strings

GL_ARB_depth_texture

Status

Complete. Approved by ARB on February 14, 2002.

Version

Last Modified Date: 13 May 2004

Number

ARB Extension #22

Dependencies

OpenGL 1.1 is required.
This extension is written against the OpenGL 1.3 Specification.

Overview

This is a clarification of the GL_SGIX_depth_texture extension. The original overview follows:

This extension defines a new depth texture format. An important application of depth texture images is shadow casting, but separating this from the shadow extension allows for the potential use of depth textures in other applications such as image-based rendering or displacement mapping. This extension does not define new depth-texture environment functions, such as filtering or applying the depth values computed from a texture but leaves this to other extensions, such as the shadow extension.

IP Status

None.

Issues

(1) How is this extension different from GL_SGIX_depth_texture?

This extension defines support for texture border values, querying depth texel resolution, and behavior when a depth texture is bound to a texture unit that's expecting RGBA texels.

(2) What about texture borders and the border value?

Texture borders are supported. The texture border value used for depth textures is the first component of TEXTURE_BORDER_COLOR.

(3) What happens when a depth texture is currently bound but RGBA texels are expected by the texture unit?

The depth texture is treated as if it were a LUMINANCE texture. It's sometimes useful to render a depth component texture as a grayscale texture.

(4) What happens when an RGBA texture is currently bound but depth texels are expected by the texture unit?

We do texturing in the normal way for an RGBA texture.

(5) What about 1D, 3D and cube maps textures? Should depth textures be supported?

RESOLVED: For 1D textures, yes, for orthogonality. For 3D and cube map textures, no. In both cases, the R coordinate that would be ordinarily be used for a shadow comparison is needed for texture lookup and won't contain a useful value. In theory, the shadow functionality could be extended to provide useful behavior for such targets, but this enhancement is left to a future extension.

(6) Why "depth" textures instead of a generic, extended-precision, single-channel texture format?

RESOLVED: We need a depth format so that glCopyTex[Sub]Image() can copy data from the depth buffer to the texture memory.

(7) Is there any particular reason that depth textures should only be used as LUMINANCE textures?

RESOLVED: Add DEPTH_TEXTURE_MODE to allow depth textures to be used as LUMINANCE, INTENSITY or ALPHA textures.

(8) It is very unlikely that depth textures when used as LUMINANCE, INTENSITY or ALPHA textures are used at their full storage precision. Should there be a query for the actual number of bits used for depth textures?

RESOLVED: No. OpenGL does not have queries for internal precision. Instead of adding it randomly for one feature, it should be looked in the broader context of providing it for more features.

(9) How should GetTexImage work for depth textures?

RESOLVED: Since GetTexImage is modeled on ReadPixels, reading depth components should require the DEPTH_COMPONENT format. Specifying a color format when querying a texture image with a DEPTH_COMPONENT base internal format should be an invalid operation. Likewise, specifying a DEPTH_COMPONENT format when querying a texture image with a color internal format should be an invalid operation. This is not only consistent with ReadPixels but how the EXT_paletted_texture and NV_texture_shader extensions amend GetTexImage to return non-color texture image data.

New Procedures and Functions

    None

New Tokens

    Accepted by the <internalFormat> parameter of TexImage1D, TexImage2D,
    CopyTexImage1D and CopyTexImage2D:

```
    DEPTH_COMPONENT
    DEPTH_COMPONENT16_ARB       0x81A5  (same as DEPTH_COMPONENT16_SGIX)
    DEPTH_COMPONENT24_ARB       0x81A6  (same as DEPTH_COMPONENT24_SGIX)
    DEPTH_COMPONENT32_ARB       0x81A7  (same as DEPTH_COMPONENT32_SGIX)
```

    Accepted by the <format> parameter of GetTexImage, TexImage1D,
    TexImage2D, TexSubImage1D, and TexSubImage2D:

```
    DEPTH_COMPONENT
```

    Accepted by the <pname> parameter of GetTexLevelParameterfv and
    GetTexLevelParameteriv:

```
    TEXTURE_DEPTH_SIZE_ARB      0x884A
```

    Accepted by the <pname> parameter of TexParameterf, TexParameteri,
    TexParameterfv, TexParameteriv, GetTexParameterfv, and GetTexParameteriv:

```
    DEPTH_TEXTURE_MODE_ARB      0x884B
```

Additions to Chapter 2 of the 1.3 Specification (OpenGL Operation)

    None

Additions to Chapter 3 of the 1.3 Specification (Rasterization)

    Section 3.8.1, Texture Image Specification, p. 116, change last
    sentence of first paragraph to:

        "The format STENCIL_INDEX is not allowed."

    Section 3.8.1, Texture Image Specification, p. 116, change final
    paragraph to read:

        "The selected groups are processed exactly as for DrawPixels, stopping
        just before final conversion. Each R, G, B, A or depth component (D)
        value so generated is clamped to [0,1]."

    Section 3.8.1, Texture Image Specification, p. 117, modify beginning of
    the first paragraph:

        "Components are then selected from the resulting R, G, B, A, or D
        values to obtain a texture with the base internal format specified
        by..."

Section 3.8.1, Texture Image Specification, p. 117, add two new paragraphs after the beginning of the first paragraph:

"Textures with a base internal format of DEPTH_COMPONENT are supported by texture image specification commands only if <target> is TEXTURE_1D, TEXTURE_2D, PROXY_TEXTURE_1D or PROXY_TEXTURE_2D. Using this format in conjunction with any other <target> will result in an INVALID_OPERATION error."

"Textures with a base internal format of DEPTH_COMPONENT require depth component data; textures with other base internal formats require RGBA component data. The error INVALID_OPERATION is generated if the base internal format is DEPTH_COMPONENT and format is not DEPTH_COMPONENT, or if the base internal format is not DEPTH_COMPONENT and format is DEPTH_COMPONENT."

Section 3.8.1, Texture Image Specification, p. 117, modify the last paragraph, which flows to p. 118:

"... If a sized internal format is specified, the mapping of the R, G, B, A, and D values to texture components is equivalent to ..."

(on p. 118) "... If a compressed internal format is specified, the mapping of the R, G, B, A, and D values to texture components is equivalent to..."

Section 3.8.1, Texture Image Specification, p. 118, add a new row to Table 3.15.

Base Internal Format	RGBA Values	Internal Components
DEPTH_COMPONENT	D	D

Section 3.8.1, Texture Image Specification, p. 118, add three new rows and one new column to Table 3.16.

Sized Internal Format	Base Int. Format	...	D bits
DEPTH_COMPONENT16_ARB	DEPTH_COMPONENT		16
DEPTH_COMPONENT24_ARB	DEPTH_COMPONENT		24
DEPTH_COMPONENT32_ARB	DEPTH_COMPONENT		32

Section 3.8.2, Alternate Texture Image Specification Commands, p. 125, modify first paragraph to read:

... "The image is taken from the framebuffer exactly as if these arguments were passed to CopyPixels, with argument <type> set to COLOR or DEPTH_COMPONENT, depending on <internalformat>, stopping after pixel transfer processing is complete. RGBA data is taken from the current color buffer while depth component data is taken from the depth buffer. If no depth buffer is present, the error INVALID_OPERATION is generated. Subsequent processing is identical to that described for TexImage2D, beginning with clamping of the R, G, B, A, or depth values from the resulting pixel groups." ...

    Section 3.8.4, Texture Parameters, p. 133, append table 3.19 with the
    following:

Name	Type	Legal Values
DEPTH_TEXTURE_MODE_ARB	enum	LUMINANCE, INTENSITY, ALPHA

    Before current section 3.8.5, Texture Wrap Modes, p. 134, insert the
    following new section. Renumber subsections of 3.8 appropriately.

        "3.8.5 Depth Component Textures

        Depth textures can be treated as LUMINANCE, INTENSITY or ALPHA
        textures during texture filtering and application. Initially,
        depth textures are interpreted as LUMINANCE."

    Modify section 3.8.7, Texture Minification, p. 139. Modify the last
    paragraph before the "Mipmapping" section to read:

        "If any of the selected tauijk, tauij, or taui in the above
        equations refer to a border texel with i < -bs, j < bs, k < -bs,
         i >= ws-bs, j >= hs-bs, or k >= ds-bs, then the border values
        given by the current setting of TEXTURE_BORDER_COLOR is used
        instead of the unspecified value or values. If the texture
        contains color components, the components of the
        TEXTURE_BORDER_COLOR vector are interpreted as an RGBA color
        to match the texture's internal format in a manner consistent
        with table 3.15. If the texture contains depth components,
        the R component of the TEXTURE_BORDER_COLOR vector is
        interpreted as the depth component value."

Additions to Chapter 4 of the 1.3 Specification (Per-Fragment Operations and the Frame Buffer)

    None

Additions to Chapter 5 of the 1.3 Specification (Special Functions)

    None

Additions to Chapter 6 of the 1.3 Specification (State and State Requests)

    Section 6.1.3, Enumerated Queries, p. 200, edit paragraph two as follows:

        ..."Queries of TEXTURE_RED_SIZE, TEXTURE_GREEN_SIZE,
        TEXTURE_BLUE_SIZE, TEXTURE_ALPHA_SIZE, TEXTURE_LUMINANCE_SIZE,
        TEXTURE_INTENSITY_SIZE, and TEXTURE_DEPTH_SIZE_ARB return the
        actual resolutions of the stored image array components, not
        the resolutions specified when the image array was defined.

    Section 6.1.4, Texture Queries, p. 201, replace the sentence two of
    paragraph two as follows:

        "Calling GetTexImage with a color format when the internal
        format of the texture image is not a color format causes the error
        INVALID_OPERATION. Likewise, calling GetTexImage with a format
        of GL_DEPTH_COMPONENT when the internal format of the texture

        image is not a depth format cause the error INVALID_OPERATION.
        If the internal format of the texture image level is a color
        format (one of RED, GREEN, BLUE, ALPHA, RGB, RGBA, LUMINANCE, or
        LUMINANCE_ALPHA), then the components are assigned among R, G,
        B, and A according to Table 6.1, starting with the first group
        in the first row, and continuing by obtaining groups in order
        from each row and proceeding from the first row to the last, and
        from the first image to the last for three-dimensional textures.
        If the internal format of the texture image level is a depth
        format (DEPTH_COMPONENT), then each depth component is assigned
        with the same ordering of rows and images."

Replace the last sentence of paragraph four with:

"Calling GetTexImage with format of COLOR_INDEX or STENCIL_INDEX causes the error INVALID_ENUM."

Section 1.6.7, Color Table Query, p.203, replace sentence two of paragraph one with:

"format and type accept the same values as do the corresponding parameters of GetTexImage except that a format of DEPTH_COMPONENT causes the error INVALID_ENUM."

Section 1.6.8, Convolution Query, p.204, replace sentence two of paragraph one with:

"format and type accept the same values as do the corresponding parameters of GetTexImage except that a format of DEPTH_COMPONENT causes the error INVALID_ENUM."

Section 1.6.9, Histogram Query, p.205, replace sentence two of paragraph one with:

"format and type accept the same values as do the corresponding parameters of GetTexImage except that a format of DEPTH_COMPONENT causes the error INVALID_ENUM."

Section 1.6.10, Minmax Query, p.205, replace sentence two of paragraph one with:

"format and type accept the same values as do the corresponding parameters of GetTexImage except that a format of DEPTH_COMPONENT causes the error INVALID_ENUM."

Additions to the GLX Specification

None

Errors

INVALID_OPERATION is generated by TexImage2D or CopyTexImage2D if <target> is not TEXTURE_2D or PROXY_TEXTURE_2D and <internalFormat> is DEPTH_COMPONENT, DEPTH_COMPONENT16_ARB, DEPTH_COMPONENT24_ARB, or DEPTH_COMPONENT32_ARB.

```
    INVALID_OPERATION is generated by TexImage1D or CopyTexImage1D if
    <target> is not TEXTURE_1D or PROXY_TEXTURE_1D and <internalFormat>
    is DEPTH_COMPONENT, DEPTH_COMPONENT16_ARB, DEPTH_COMPONENT24_ARB, or
    DEPTH_COMPONENT32_ARB.

    INVALID_OPERATION is generated by TexImage1D or TexImage2D if <format>
    is DEPTH_COMPONENT and <internalFormat> is not DEPTH_COMPONENT,
    DEPTH_COMPONENT16_ARB, DEPTH_COMPONENT24_ARB, or DEPTH_COMPONENT32_ARB.

    INVALID_OPERATION is generated by TexImage1D or TexImage2D if
    <internalFormat> is DEPTH_COMPONENT, DEPTH_COMPONENT16_ARB,
    DEPTH_COMPONENT24_ARB, or DEPTH_COMPONENT32_ARB, and <format> is not
    DEPTH_COMPONENT.

    INVALID_OPERATION is generated by TexSubImage1D or TexSubImage2D if
    <format> is DEPTH_COMPONENT and the base internal format of the
    texture is not DEPTH_COMPONENT, DEPTH_COMPONENT16_ARB,
    DEPTH_COMPONENT24_ARB, or DEPTH_COMPONENT32_ARB.

    INVALID_OPERATION is generated by TexSubImage1D or TexSubImage2D if
    <format> is not DEPTH_COMPONENT and the base internal format of
    the texture is DEPTH_COMPONENT, DEPTH_COMPONENT16_ARB,
    DEPTH_COMPONENT24_ARB, or DEPTH_COMPONENT32_ARB.

    INVALID_OPERATION is generated by TexImage3D if <internalFormat>
    is DEPTH_COMPONENT, DEPTH_COMPONENT16_ARB, DEPTH_COMPONENT24_ARB,
    or DEPTH_COMPONENT32_ARB.

    INVALID_OPERATION is generated by CopyTexImage1D or CopyTexImage2D if
    <internalFormat> is DEPTH_COMPONENT, DEPTH_COMPONENT16_ARB,
    DEPTH_COMPONENT24_ARB, or DEPTH_COMPONENT32_ARB, and there is no depth
    buffer.

    INVALID_OPERATION is generated by CopyTexSubImage1D or CopyTexSubImage2D
    if the base internal format of the texture is DEPTH_COMPONENT and there
    is no depth buffer.

    INVALID_ENUM is generated if TexParameter[if] parameter <pname>
    is DEPTH_TEXTURE_MODE_ARB and parameter <param> is not ALPHA,
    LUMINANCE, or INTENSITY.

    INVALID_OPERATION is generated if GetTexImage parameter <format>
    is one of RED, GREEN, BLUE, ALPHA, RGB, RGBA, BGR, BGRA, LUMINANCE,
    or LUMINANCE_ALPHA but the internal format of the texture level
    image is not a color format.

    INVALID_OPERATION is generated if GetTexImage parameter <format>
    is DEPTH_COMPONENT but the internal format of the texture level
    image is not a depth format.

    Eliminate the INVALID_ENUM generated if GetTexImage parameter
    <format> is DEPTH_COMPONENT.  (but this should still be an error for
    GetColorTable, GetConvolutionFilter, GetHistogram, and GetMinmax).
```

New State

In table 6.12, Texture Objects, p. 202, add the following:

Get Value	Type	Get Command	Initial Value	Description	Sec.	Attribute
TEXTURE_DEPTH_SIZE	Z+	GetTexLevelParameter	0	xD texture image i's depth resolution	3.8	-

In table 6.16, Texture Objects, p. 224, add the following:

Get Value	Type	Get Command	Initial Value	Description	Sec.	Attribute
DEPTH_TEXTURE_MODE_ARB	Z_3	GetTexParameter[if]v	LUMINANCE	depth texture mode	3.8.13	texture

New Implementation Dependent State

None

Revision History

```
    19 March 2001
        - initial version
    22 June 2001
        - added 1D textures to issue 4
    16 November 2001
        - removed TEXTURE_BORDER_DEPTH.  use the first component of
          TEXTURE_BORDER_COLOR to specify the depth border value.
        - Added new language in section 3.8.5 to describe how
          TEXTURE_BORDER_COLOR is used with depth textures.
        - Inserted new issue item #4.
    17 November 2001
        - Changed issue 4 resolution.
        - Rewrote section 3.8.4
    12 December 2001 (Pat Brown)
        - Retargeted against the OpenGL 1.3 specification.
        - Depth textures are allowed only on 1D and 2D targets.  Shadowing is
          problematic for 3D and cube map textures.
        - Updated base and sized internal format tables.
        - Documented a couple missing error conditions for TexImage and
          TexSubImage calls where <format> and the texture internal format are
          incompatible.
        - Minor cleanups to provide for depth components in wording that
          formerly assumed RGBA components only.
    13 December 2001
        - Removed a few lingering references to glTexImage3D.
        - Rewrite the first and last error conditions to be clearer.
        - replace "1.2" with "1.3" in a few places.
    - fixed a few more error conditions (Pat Brown)
    11 January 2002
        - fixed "intented" typo
        - added sentence saying that TEXTURE_LUMINANCE_SIZE may be used
          to query the effective resolution of a depth textures when it's
          interpreted as a luminance texture.
    18 January 2002
        - Allow depth textures to be used as LUMINANCE, INTENSITY or ALPHA
          textures (Bimal Poddar)
```

```
21 January 2002
    - Added issue #8 to deal with actual depth texture precision.
      Fixed error to be INVALID_ENUM instead of INVALID_OPERATION.
13 May 2004 (mjk)
    - Document GetTexImage behavior when depth texture formats are
      supported.
    - Document that GetColorTable, GetConvolutionFilter, GetHistogram,
      and GetMinmax now differ from GetTexImage in that
      DEPTH_COMPONENT is (still) not a legal format for these
      queries.
    - Document in "New Tokens" that DEPTH_COMPONENT is a newly accepted
      token for the <internalFormat> parameter of TexImage1D,
      TexImage2D, CopyTexImage1D and CopyTexImage2D; and the <format>
      parameter of GetTexImage, TexImage1D, TexImage2D, TexSubImage1D,
      and TexSubImage2D.
    - Fix mangled sentence in 3.8.5
```

Name

    ARB_draw_buffers

Name Strings

    GL_ARB_draw_buffers

Contributors

    Benj Lipchak, ATI
    Bill Licea-Kane, ATI

Contact

    Rob Mace, ATI Research (mace 'at' ati.com)

IP Status

    No known IP issues.

Status

    Complete. Approved by the ARB on July 23, 2004.

Version

    Last Modified Date: July 26, 2004
    Revision: 14

Number

    ARB Extension #37

Dependencies

    The extension is written against the OpenGL 1.5 Specification.

    OpenGL 1.3 is required.

    ARB_fragment_program affects the definition of this extension.

    ARB_fragment_shader affects the definition of this extension.

Overview

    This extension extends ARB_fragment_program and ARB_fragment_shader
    to allow multiple output colors, and provides a mechanism for
    directing those outputs to multiple color buffers.

Issues

(1) How many GL_DRAW_BUFFER#_ARB enums should be reserved?

      RESOLVED: We only need 4 currently, but for future expandability
      it would be nice to keep the enums in sequence. We'll specify
      16 for now, which will be more than enough for a long time.

```
    (2) How should multisample work when there are multiple output
        colors being rendered to multiple draw buffers?

      Basic options are:
        (a) Color 0 is written to the multisample buffer and then the
            multisample buffer is resolved to all the color buffers.
            This option would be consistent with GL's idea of a single
            multisample buffer, but would be really useless and defeat
            the purpose of multiple output colors.
        (b) Have a separate multisample color buffer for each output
            color/draw buffer.  This would be useful but would all
            implementations be able to handle it?
        (c) Don't allow multiple output colors and multisampling to
            be combined by restricting MAX_DRAW_BUFFERS_ARB to 1
            for contexts with multisample buffers.  This is simple
            and would allow a future extension to allow (b).

      RESOLUTION: (b) and (c).  Samples will contain separate color
      values for each output color.  Implementations that can not
      support this can restrict MAX_DRAW_BUFFERS_ARB to 1 for contexts
      with multisample buffers.

    (3) Should gl_FragColor be aliased to gl_FragData[0]?

      RESOLUTION: No.  A shader should write either gl_FragColor, or
      gl_FragData[n], but not both.

      Writing to gl_FragColor will write to all draw buffers specified
      with DrawBuffersARB.

    (4) Should gl_FragData[n] be clamped?

      RESOLUTION: They will be clamped if fragment color clamping is
      enabled.
```

New Procedures and Functions

```
    void DrawBuffersARB(sizei n, const enum *bufs);
```

New Tokens

```
    Accepted by the <pname> parameters of GetIntegerv, GetFloatv,
    and GetDoublev:

        MAX_DRAW_BUFFERS_ARB                        0x8824
        DRAW_BUFFER0_ARB                            0x8825
        DRAW_BUFFER1_ARB                            0x8826
        DRAW_BUFFER2_ARB                            0x8827
        DRAW_BUFFER3_ARB                            0x8828
        DRAW_BUFFER4_ARB                            0x8829
        DRAW_BUFFER5_ARB                            0x882A
        DRAW_BUFFER6_ARB                            0x882B
        DRAW_BUFFER7_ARB                            0x882C
        DRAW_BUFFER8_ARB                            0x882D
        DRAW_BUFFER9_ARB                            0x882E
        DRAW_BUFFER10_ARB                           0x882F
```

DRAW_BUFFER11_ARB	0x8830
DRAW_BUFFER12_ARB	0x8831
DRAW_BUFFER13_ARB	0x8832
DRAW_BUFFER14_ARB	0x8833
DRAW_BUFFER15_ARB	0x8834

Additions to Chapter 2 of the OpenGL 1.5 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the OpenGL 1.5 Specification (Rasterization)

Modify Section 3.2.1, Multisampling (p. 71)

(replace the second paragraph with)

An additional buffer, called the multisample buffer, is added to the framebuffer. Pixel sample values, including color, depth, and stencil values, are stored in this buffer. Samples contain separate color values for each output color. When the framebuffer includes a multisample buffer, it does not include depth or stencil buffers, even if the multisample buffer does not store depth or stencil values. Color buffers (left, right, front, back, and aux) do coexist with the multisample buffer, however.

Modify Section 3.11.2, Fragment Program Grammar and Semantic Restrictions (ARB_fragment_program)

(replace <resultBinding> grammar rule with these rules)

```
<resultBinding>        ::= "result" "." "color" <optOutputColorNum>
                         | "result" "." "depth"

<optOutputColorNum>    ::= ""
                         | "[" <outputColorNum> "]"

<outputColorNum>       ::= <integer> from 0 to MAX_DRAW_BUFFERS_ARB-1
```

Modify Section 3.11.3.4, Fragment Program Results

(modify Table X.3)

Binding	Components	Description
result.color[n]	(r,g,b,a)	color n
result.depth	(*,*,*,d)	depth coordinate

Table X.3: Fragment Result Variable Bindings. Components labeled "*" are unused. "[n]" is optional -- color <n> is used if specified; color 0 is used otherwise.

(modify third paragraph) If a result variable binding matches "result.color[n]", updates to the "x", "y", "z", and "w" components of the result variable modify the "r", "g", "b", and "a" components, respectively, of the fragment's corresponding output color. If "result.color[n]" is not both bound by the fragment program and written by some instruction of the program, the output color <n> of the fragment program is undefined.

Add a new Section 3.11.4.5.3 (ARB_fragment_program)

3.11.4.5.3 Draw Buffers Program Option

If a fragment program specifies the "ARB_draw_buffers" option, it will generate multiple output colors, and the result binding "result.color[n]" is allowed, as described in section 3.11.3.4, and with modified grammar rules as set forth in section 3.11.2. If this option is not specified, a fragment program that attempts to bind "result.color[n]" will fail to load, and only "result.color" will be allowed.

Add a new section 3.11.6 (ARB_fragment_shader)

Section 3.11.6 Fragment Shader Output

The OpenGL Shading Language specification describes the values that may be output by a fragment shader. These are gl_FragColor, gl_FragData[n], and gl_FragDepth. If fragment color clamping is enabled, the final fragment color values or the final fragment data values written by a fragment shader are clamped to the range [0,1] and then converted to fixed-point as described in section 2.13.9, Final Color Processing.

The final fragment depth written by a fragment shader is first clamped to [0,1] then converted to fixed-point as if it were a window z value. See Section 2.10.1, Controlling the Viewport. Note that the depth range computation is NOT applied here, only the conversion to fixed-point.

The OpenGL Shading Language specification defines what happens when color and/or depth are not written. Those rules are repeated here.

Writing to gl_FragColor specifies the fragment color that will be used by the subsequent fixed functionality pipeline. If subsequent fixed functionality consumes fragment color and an execution of a fragment shader does not write a value to gl_FragColor then the fragment color consumed is undefined.

Writing to gl_FragData[n] specifies the fragment data that will be used by the subsequent fixed functionality pipeline. If subsequent fixed functionality consumes fragment data and an execution of a fragment shader does not write a value to gl_FragData[n] then the fragment data consumed is undefined.

If a shader statically assigns a value to gl_FragColor, it may not assign a value to gl_FragData[n]. If a shader statically writes a value to gl_FragData[n], it may not assign a value to gl_FragColor. That is, a shader may assign values to either gl_FragColor or gl_FragData[n], but not both.

Writing to gl_FragDepth will establish the depth value for the fragment being processed. If depth buffering is enabled, and a shader does not write gl_FragDepth, then the fixed function value for depth will be used as the fragment's depth value. If a shader statically assigns a value to gl_FragDepth, and there is an execution path through the shader that does not set gl_FragDepth, then the value of the fragment's depth may be undefined for some executions of the shader. That is, if a shader statically writes gl_FragDepth, then it is responsible for always writing it.

Note, statically assigning a value to gl_FragColor, gl_FragData[n] or gl_FragDepth means that there is a line of code in the fragment shader source that writes a value to gl_FragColor, gl_FragData[n] or gl_FragDepth, respectively, even if that line of code is never executed.

Additions to Chapter 4 of the OpenGL 1.5 Specification (Per-Fragment Operations and the Frame Buffer)

Replace Section 4.2.1, Selecting a Buffer for Writing (p. 183)

4.2.1 Selecting Color Buffers for Writing

The first such operation is controlling the color buffers into which each of the output colors are written. This is accomplished with either DrawBuffer or DrawBuffersARB.

The command

```
  void DrawBuffer(enum buf);
```

defines the set of color buffers to which output color 0 is written. <buf> is a symbolic constant specifying zero, one, two, or four buffers for writing. The constants are NONE, FRONT_LEFT, FRONT_RIGHT, BACK_LEFT, BACK_RIGHT, FRONT, BACK, LEFT, RIGHT, FRONT_AND_BACK, and AUX0 through AUXn, where n + 1 is the number of available auxiliary buffers.

The constants refer to the four potentially visible buffers front left, front right, back left, and back right, and to the auxiliary buffers. Arguments other than AUXi that omit reference to LEFT or RIGHT refer to both left and right buffers. Arguments other than AUXi that omit reference to FRONT or BACK refer to both front and back buffers. AUXi enables drawing only to auxiliary buffer i. Each AUXi adheres to AUXi = AUX0 + i. The constants and the buffers they indicate are summarized in Table 4.3. If DrawBuffer is supplied with a constant (other than NONE) that does not indicate any of the color buffers allocated to the GL context, the error INVALID_OPERATION results.

symbolic constant	front left	front right	back left	back right	aux i
NONE					
FRONT_LEFT	*				
FRONT_RIGHT		*			
BACK_LEFT			*		
BACK_RIGHT				*	
FRONT	*	*			

BACK			*	*	
LEFT	*		*		
RIGHT		*		*	
FRONT_AND_BACK	*	*	*	*	
AUXi					*

Table 4.3: Arguments to DrawBuffer and the buffers that they indicate.

DrawBuffer will set the draw buffer for output colors other than 0 to NONE.

The command

```
void DrawBuffersARB(sizei n, const enum *bufs);
```

defines the draw buffers to which all output colors are written. <n> specifies the number of buffers in <bufs>. <bufs> is a pointer to an array of symbolic constants specifying the buffer to which each output color is written. The constants may be NONE, FRONT_LEFT, FRONT_RIGHT, BACK_LEFT, BACK_RIGHT, and AUX0 through AUXn, where n + 1 is the number of available auxiliary buffers. The draw buffers being defined correspond in order to the respective output colors. The draw buffer for output colors beyond <n> is set to NONE.

Except for NONE, a buffer should not appear more then once in the array pointed to by <bufs>. Specifying a buffer more then once will result in the error INVALID_OPERATION.

If a fragment program is not using the "ARB_draw_buffers" option, DrawBuffersARB specifies a set of draw buffers into which output color 0 is written.

If a fragment shader writes to "gl_FragColor", DrawBuffersARB specifies a set of draw buffers into which the color written to "gl_FragColor" is written.

The maximum number of draw buffers is implementation dependent and must be at least 1. The number of draw buffers supported can be queried with the state MAX_DRAW_BUFFERS_ARB.

The constants FRONT, BACK, LEFT, RIGHT, and FRONT_AND_BACK that refer to multiple buffers are not valid for use in DrawBuffersARB and will result in the error INVALID_OPERATION.

If DrawBuffersARB is supplied with a constant (other than NONE) that does not indicate any of the color buffers allocated to the GL context, the error INVALID_OPERATION will be generated. If <n> is greater than MAX_DRAW_BUFFERS_ARB, the error INVALID_OPERATION will be generated.

Indicating a buffer or buffers using DrawBuffer or DrawBuffersARB causes subsequent pixel color value writes to affect the indicated buffers. If more than one color buffer is selected for drawing, blending and logical operations are computed and applied independently for each buffer. If there are multiple output colors being written to multiple buffers, the alpha used in alpha to coverage and alpha test is the alpha of output color 0.

Specifying NONE as the draw buffer for an output color will inhibit that output color from being written to any buffer.

Monoscopic contexts include only left buffers, while stereoscopic contexts include both left and right buffers. Likewise, single buffered contexts include only front buffers, while double buffered contexts include both front and back buffers. The type of context is selected at GL initialization.

The state required to handle color buffer selection is an integer for each supported output color. In the initial state, draw buffer for output color 0 is FRONT if there are no back buffers; otherwise it is BACK. The initial state of draw buffers for output colors other then 0 is NONE.

Additions to Chapter 5 of the OpenGL 1.5 Specification (Special Functions)

None

Additions to Chapter 6 of the OpenGL 1.5 Specification (State and State Requests)

None

Additions to Chapter 3 of the OpenGL Shading Language 1.10 Specification (Basics)

Add a new Section 3.3.1, GL_ARB_draw_buffers Extension (p. 13)

3.3.1 GL_ARB_draw_buffers Extension

To use the GL_ARB_draw_buffers extension in a shader it must be enabled using the #extension directive.

The shading language preprocessor #define GL_ARB_draw_buffers will be defined to 1, if the GL_ARB_draw_buffers extension is supported.

Dependencies on ARB_fragment_program

If ARB_fragment_program is not supported then all changes to section 3.11 of ARB_fragment_program and the fragment program specific part of section 4.2.1 are removed.

Dependencies on ARB_fragment_shader

If ARB_fragment_shader is not supported then all changes to section 3.11 of ARB_fragment_shader, section 3.3.1 of the Shading Language Specification, and the fragment shader specific part of section 4.2.1 are removed.

Interactions with possible future extensions

If there is some other future extension that defines multiple

    color outputs then this extension and glDrawBuffersARB could be
    used to define the destinations for those outputs. This extension
    need not be used only with ARB_fragment_program.

Errors

    The error INVALID_OPERATION is generated by DrawBuffersARB if a
    color buffer not currently allocated to the GL context is specified.

    The error INVALID_OPERATION is generated by DrawBuffersARB if <n>
    is greater than the state MAX_DRAW_BUFFERS_ARB.

    The error INVALID_OPERATION is generated by DrawBuffersARB if value
    in <bufs> does not correspond to one of the allowed buffers.

    The error INVALID_OPERATION is generated by DrawBuffersARB if a draw
    buffer other then NONE is specified more then once in <bufs>.

New State

    (table 6.19, p227) add the following entry:

Get Value	Type	Get Command	Initial Value	Description	Section	Attribute
DRAW_BUFFERi_ARB	Z10*	GetIntegerv	see 4.2.1	Draw buffer selected for output color i	4.2.1	color-buffer

New Implementation Dependent State

Get Value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
MAX_DRAW_BUFFERS_ARB	Z+	GetIntegerv	1	Maximum number of active draw buffers	4.2.1	-

Revision History

```
   Date: 7/26/2004
   Revision: 14
      - Clarified interaction of gl_FragColor and multiple draw buffers.
      - Updated dependencies section.
      - Added real ARB extension #.

   Date: 7/22/2004
   Revision: 13
      - Converted from ATI_draw_buffers to ARB_draw_buffers.

   Date: 7/21/2004
   Revision: 12
      - Updated intro to mention ARB_fragment_shader.
      - Marked which sections modify ARB_fragment_program and
        ARB_fragment_shader.
      - Added "Dependencies on ARB_fragment_shader".
      - Added extension section 3.3.1 to Shading Language spec.
      - Resolved interaction with multisample (issue 2).
      - Fixed typos.

   Date: 6/9/2004
```

```
    Revision: 11
       - Added GLSL integration.

    Date: 4/27/2004
    Revision: 10
       - Replaced modification to section 4.2.1 with a complete
         replacement for the section, the individual modifications were
         getting too cumbersome.
       - Added issue (2) on multisampling.

    Date: 4/15/2004
    Revision: 9
       - Specified that it is the alpha of color 0 that is used for alpha
         test.

    Date: 12/30/2002
    Revision: 8
       - Clarified that DrawBuffersATI will set the set of draw buffers
         to write color output 0 to when the "ATI_draw_buffer" fragments
         program option is not in use.

    Date: 9/27/2002
    Revision: 7
       - Fixed confusion between meaning of color buffer and draw buffer
         in last revision.
       - Fixed mistake in when an error is generated based on the <n>
         argument of DrawBuffersATI.

    Date: 9/26/2002
    Revision: 6
       - Cleaned up and put in sync with latest ARB_fragment_program
         revision (#22).  Some meaningless changes made just in the name
         of consistency.

    Date: 9/11/2002
    Revision: 5
       - Added section 3.11.4.5.3.
       - Added enum numbers to New Tokens.

    Date: 9/9/2002
    Revision: 4
       - Changed error from MAX_OUTPUT_COLORS to MAX_DRAW_BUFFERS_ATI.
       - Changed 3.10 section numbers to 3.11 to match change to
         ARB_fragment_program spec.
       - Changed ARB_fragment_program from required to affects, and
         added section on interactions with it and future extensions
         that define multiple color outputs.

    Date: 9/6/2002
    Revision: 3
       - Changed error to INVALID OPERATION.
       - Cleaned up typos.

    Date: 8/19/2002
    Revision: 2
       - Added a paragraph that specifically points out that the
         constants that refer to multiple buffers are not allowed with
```

```
          DrawBuffersATI.
        - Changed bufs to <bufs> in a couple of places.

    Date: 8/16/2002
    Revision: 1
        - First draft for circulation.
```

Name

    ARB_fragment_program

Name Strings

    GL_ARB_fragment_program

IP Status

    Microsoft claims to own intellectual property related to this
    extension.

Status

    Complete. Approved by ARB on September 18, 2002

Version

    Last Modified Date: August 22, 2003
    Revision: 26

Number

    ARB Extension #27

Dependencies

    The extension is written against the OpenGL 1.3 Specification.

    OpenGL 1.3 is required.

    EXT_texture_lod_bias or OpenGL 1.4 is required.

    OpenGL 1.4 affects the definition of this extension.

    ARB_vertex_blend and EXT_vertex_weighting affect the definition of
    this extension.

    ARB_matrix_palette affects the definition of this extension.

    ARB_transpose_matrix affects the definition of this extension.

    EXT_fog_coord affects the definition of this extension.

    EXT_texture_rectangle affects the definition of this extension.

    ARB_shadow interacts with this extension.

    ARB_vertex_program interacts with this extension.

    ATI_fragment_shader interacts with this extension.

    NV_fragment_program interacts with this extension.

Overview

Unextended OpenGL mandates a certain set of configurable per-fragment computations defining texture application, texture environment, color sum, and fog operations. Several extensions have added further per-fragment computations to OpenGL. For example, extensions have defined new texture environment capabilities (ARB_texture_env_add, ARB_texture_env_combine, ARB_texture_env_dot3, ARB_texture_env_crossbar), per-fragment depth comparisons (ARB_depth_texture, ARB_shadow, ARB_shadow_ambient, EXT_shadow_funcs), per-fragment lighting (EXT_fragment_lighting, EXT_light_texture), and environment mapped bump mapping (ATI_envmap_bumpmap).

Each such extension adds a small set of relatively inflexible per-fragment computations.

This inflexibility is in contrast to the typical flexibility provided by the underlying programmable floating point engines (whether micro-coded fragment engines, DSPs, or CPUs) that are traditionally used to implement OpenGL's texturing computations. The purpose of this extension is to expose to the OpenGL application writer a significant degree of per-fragment programmability for computing fragment parameters.

For the purposes of discussing this extension, a fragment program is a sequence of floating-point 4-component vector operations that determines how a set of program parameters (not specific to an individual fragment) and an input set of per-fragment parameters are transformed to a set of per-fragment result parameters.

The per-fragment computations for standard OpenGL given a particular set of texture and fog application modes (along with any state for extensions defining per-fragment computations) is, in essence, a fragment program. However, the sequence of operations is defined implicitly by the current OpenGL state settings rather than defined explicitly as a sequence of instructions.

This extension provides an explicit mechanism for defining fragment program instruction sequences for application-defined fragment programs. In order to define such fragment programs, this extension defines a fragment programming model including a floating-point 4-component vector instruction set and a relatively large set of floating-point 4-component registers.

The extension's fragment programming model is designed for efficient hardware implementation and to support a wide variety of fragment programs. By design, the entire set of existing fragment programs defined by existing OpenGL per-fragment computation extensions can be implemented using the extension's fragment programming model.

Issues

This extension is closely related to ARB_vertex_program, and is in sync with revision 36 of that spec. ARB_fragment_program will continue to track changes made to ARB_vertex_program.

(1) Should we provide precision queries?

RESOLVED: We've decided not to include precision queries. Implementations are expected to meet or exceed the precision guidelines set forth in the core GL spec, section 2.1.1, p. 6, as ammended by this extension.

To summarize section 2.1.1, the maximum representable magnitude of colors must be at least 2^10, while the maximum representable magnitude of other floating-point values must be at least 2^32. The individual results of floating-point operations must be accurate to about 1 part in 10^5.

Here are the reasons why precision queries were not included:

1. It is unclear what the queries should be:
   a) min, max, [0,1) granularity
   b) min +, max +, min -, max -, [0,1) granularity
   c) IEEE mantissa bits, IEEE exponent bits
2. Due to instruction emulation, there is no way to query the actual precision that can be expected. Should the query return the best-case or worst-case precision?
3. Implementations may support multiple precisions, on a per-instruction basis or across the board. How would this be exposed?
4. Current implementations are able to meet the minimum requirements specified in the core GL, thanks to its sufficiently loose wording "... so that the individual results of floating-point operations are accurate to ABOUT 1 part in 10^5." (Emphasis added.)
5. A conformance test can act as watchdog to ensure implementations are not cutting corners on precision.
6. Adding precision queries would require a new entrypoint.

See issue 22 regarding reduced-precision modes.

(2) Should the LOD biased texture sample be optional?

RESOLVED: TXB support is mandatory. This exposes useful functionality which enables blurring and sharpening effects. It will be more useful to entirely override derivatives (scale factor) rather than just biasing the level-of-detail. This would be a future extension to fragment programs.

It should be noted here that the bias introduced per-fragment by TXB is added to any per-object or per-stage LOD bias. If per-fragment LOD bias is not necessary, using the per-object and/or per-stage LOD biases may perform better.

(3) Should we include the ability to bind to the color matrix? How about others? Program matrices?

RESOLVED: We will not specifically add anything that depends on the ARB_imaging subset. So we have not included matrix bindings to the color matrix (or parameter bindings to the color biases, etc.). However, we have included matrix binding support and support for all of the matrices present in ARB_vertex_program.

(4) Should we include the ability to bind to just a texcoord attribute's S,T components? (Or just S, or S,T,P for that matter?)

RESOLVED: No. Issue #15 below obviates this issue by making the texture coordinate usage within a program explicit, thereby making optimizations to reduce the number of interpolated texture coordinates something an implementation can do at compile time instead of having to do it during every texture target change.

(5) What other instructions should be added? Should any be removed?

RESOLVED: The differences between the ARB_vertex_program instruction set and the ARB_fragment_program instruction set are minimal. ARB_fragment_program removes the LOG and EXP rough approximation instructions and the ARL address register load instruction. ARB_fragment_program adds the SIN/COS/SCS trigonometric instructions, the LRP linear interpolation instruction, the CMP compare instruction, and the TEX/TXP/TXB/KIL texture instructions.

(6) Should depth output be a program option or a mandatory feature?

RESOLVED: Depth output capability should be mandatory.

(6a) How should per-vertex geometric depth clipping be handled when replacing depth in a fragment program?

RESOLVED: Per-vertex geometric depth clipping should be performed by the GL as usual, so no spec change is required. The ideal behavior would be to disable near and far clipping planes when replacing depth, but not all implementations can natively support disabling individual clip planes.

(6b) How should depth output from the fragment program be further processed before being handed to the per-fragment operations?

RESOLVED: Depth gets clamped by GL to [0,1]. App has access to depth range as a bindable parameter if it wants to either scale and bias its depth to fall within the depth range, or to kill fragments outside the depth range.

(7) If a fragment program does not write a color value, what should be the final color of the fragment?

RESOLVED: The final fragment color is undefined. Note that it may be perfectly reasonable to have a program that computes depth values but not colors. Fragment colors are often irrelevant if color writes are disabled (via ColorMask).

(7a) If a fragment program does not write a depth value, what should be the final depth value of the fragment?

RESOLVED: "Depth fly-over" (using the conventional depth produced by rasterization) should happen whenever a depth-replacing program is not in use. A depth-replacing program is defined as a program that writes to result.depth in at least one instruction. The presence of a depth declaration alone DOES NOT designate a depth-replacing program. The intention is that a future extension introducing conditional execution will still consider a program to be depth-replacing even if the instruction(s) writing to result.depth do(es) not execute.

Other considered definitions of depth-replacing program:

1. The presence of a depth declaration -OR- the use of result.depth as an instruction destination anywhere in the program designates a depth-replacing program.
2. Every program is a depth-replacing program, but the GL initializes the depth output to be the depth produced by rasterization. The app may then overwrite the depth output.
3. Every program is a depth-replacing program, and the app is solely responsible for copying the depth input to depth output if desired.

(8) Should relative addressing, like that defined in ARB_vertex_program, be supported in this spec?

RESOLVED: No, relative addressing won't be included in this spec.

(9) Should full-featured operand component swizzling, like that defined in ARB_vertex_program, be supported in this spec?

RESOLVED: Yes, full swizzling is mandatory.

(10) Should texture instructions contain specific limitations on operations that can be performed? For example, should write masks or operand component swizzling be disallowed?

RESOLVED: Texture instructions are specified to be very similar to ALU instructions. They have been given 3-letter names, they allow writemasking and saturation (which would be useful for floating-point texture formats), source swizzles and negates, and the ability to use parameters as sources.

(11) Should we standardize options for stencil or aux data buffer outputs?

RESOLVED: Stencil and aux data buffers will be saved for a possible future extension to fragment programs.

(12) Should depth output be pulled from the 3rd or 4th component?

RESOLVED: 3rd component, as the 3rd component is also used for depth input from the "fragment.position" attribute.

(13) Which stages are subsumed by fragment programs?

RESOLVED: Texturing, color sum, and fog.

(14) What should the minimum resource limits be?

RESOLVED: 10 attributes, 24 parameters, 4 texture indirections, 48 ALU instructions, 24 texture instructions, and 16 temporaries.

(15) OpenGL provides a hierarchy of texture enables (cube map, 3D, 2D, 1D). Should the texture sampling instructions here override that hierarchy and select specific texture targets?

RESOLVED: Yes. This removes a potential pitfall for developers: leaving the hierarchy of enables in an undesired state. It makes programs more readable as the intent of the sample is more obvious. Finally, it allows compilers to be more aggressive as to which texcoord components are "don't cares" without having to recompile programs when fixed-function texenables change. One drawback is that programs cannot be reused for both 2D and 3D texturing, for example, by simply changing the texture enables.

Texture sampling can be specified by instructions like

```
TEX myTexel, fragment.texcoord[1], texture[2], 3D;
```

which would indicate to use texture coordinate set number 1 to sample from the texture object bound to the TEXTURE_3D target on texture image unit 2.

Each texture unit can have only one "active" target. Programs are not allowed to reference different texture targets in the same texture image unit. In the example above, any other texture instructions using texture image unit 2 must specify the 3D texture target.

Note that every texture image unit always has a texture bound to every texture target, whether it is a named texture object or a default texture. However, the texture may not be complete as defined in section 3.8.9 of the core GL spec. See issue 23.

(16) Should aux texture units be additional units on top of existing full-featured texture units, or should this spec fully deprecate "legacy" texture units and only expose texture coordinate sets and texture image units?

Background: Some implementations are able to expose more "texture image units" (texture maps and associated parameters) than "texture coordinate sets" (current texcoords, texgen, and texture matrices). A conventional GL "texture unit" encompasses both a texture image unit and a texture coordinate set as well as texture environment state.

RESOLVED: Yes, deprecate "legacy" texture units. This is a more flexible model.

(17) Should fragment programs affect all fragments, or just those produced by the rasterization of points, lines, and triangles?

RESOLVED: Every fragment generated by the GL is subject to fragment program mode. This includes point, line, and polygon primitives as well as pixel rectangles and bitmaps.

(18) Should per-fragment position and fogcoord be bindable as fragment attributes?

RESOLVED: Yes, interpolated fogcoord will make per-fragment fog application possible, in addition to full fog stage subsummation. Interpolated window position, especially depth, enables interesting depth-replacing algorithms.

(19) What characters should be used to identify individual components in swizzle selectors and write masks?

RESOLVED: ARB_vertex_program provides "xyzw". This extension supports "xyzw" and also provides "rgba" for better readability when dealing with RGBA color values. Adding support for special identifiers for dealing with texture coordinates was considered and rejected. "strq" could be used to identify texture coordinate components, but the "r" would conflict with the "r" from "rgba". "stpq" would be another possibility, but could be a source of confusion.

(20) Should implementations be required to support all programs that fit within the exported limits on the number of resources (e.g., instructions, temporaries) that can be present in a program, even if it means falling back to software? Should implementations be required to reject programs that could never be accelerated?

RESOLVED: No and no. An implementation is allowed to fail ProgramStringARB due to the program exceeding native resources. Note that this failure must be invariant with respect to all other OpenGL state. In other words, a program cannot succeed to load with default state, but then fail to load when certain GL state is altered. However, an implementation is not required to fail when a program would exceed native resources, and is in fact encouraged to fallback to a software path. See issue 21 for a way of determining if this has happened.

This notable departure from ARB_vertex_program was made as an accommodation to vendors who could not justify implementing a software fallback path which would be relatively slow even compared to an ARB_vertex_program software fallback path.

Two issues with this decision:

1. The API limits become hints, and one can no longer tell by visual inspection whether or not a program will load on every implementation.
2. Program loading will now depend on the optimizer, which may vary from release to release of an implementation. A program that succeeded to load when an ISV first wrote it may fail to load in a future driver version, and vice versa.

(21) How can applications determine if their programs are too large to run on the native (likely hardware) implementation, and therefore may run with reduced performance?

RESOLVED: The following code snippet uses a native resource query to guarantee a program is loaded natively (or not at all):

```
GLboolean ProgramStringIsNative(GLenum target, GLenum format,
                                GLsizei len, const GLvoid *string)
{
    GLint errorPos, isNative;
    glProgramStringARB(target, format, len, string);
    glGetIntegerv(GL_PROGRAM_ERROR_POSITION_ARB, &errorPos);
    glGetProgramivARB(GL_FRAGMENT_PROGRAM_ARB,
        GL_PROGRAM_UNDER_NATIVE_LIMITS_ARB, &isNative);
    if ((errorPos == -1) && (isNative == 1))
        return GL_TRUE;
    else
        return GL_FALSE;
}
```

Note that a program that successfully loads, and falls under the native limits, is still not guaranteed to execute in hardware. Lack of other resources (e.g., texture memory) or the use of other OpenGL features not natively supported by the implementation (e.g., textures with borders) may also prevent the program from executing in hardware.

(22) Should we provide applications with a method to control the level of precision used to carry out fragment program computations?

RESOLVED: Yes. The GL implementation ultimately has control over the level of precision used for fragment program computations. However, the "ARB_precision_hint_fastest" and "ARB_precision_hint_nicest" program options allow applications to guide the GL implementation in its precision selection. The "fastest" option encourages the GL to minimize execution time, with possibly reduced precision. The "nicest" option encourages the GL to maximize precision, with possibly increased execution time.

If the precision hint is not "fastest", GL implementations should perform low-precision operations only if they could not appreciably affect the final results of the program. Regardless of the precision hint, GL implementations are discouraged from reducing the precision of computations so aggressively that final rendering results could be seriously compromised due to overflow of intermediate values or insufficient number of mantissa bits.

Some implementations may provide only a single level of precision, in which case these hints may have no effect. However, all implementations will accept these options, even if they are silently ignored.

More explicit control of precision, such as provided in "C" with data types such as "short", "int", "float", "double", may also be a desirable feature, but this level of detail is left to a separate extension.

(23) What is the result of a sample from an incomplete texture? The definition of texture completeness can be found in section 3.8.9 of the core GL spec.

RESOLVED: The result of a sample from an incomplete texture is the constant vector (0,0,0,1). The benefit of defining the result to be a constant is that broken apps are guaranteed to generate unexpected (black) results from their bad samples. If we were to leave the result undefined, some implementations may generate expected results some of the time, for example when magfiltering, giving app developers a false sense of correctness in their apps.

(24) What is a texture indirection, and how is it counted?

RESOLVED: On some implementations, fragment programs that have complex texture dependency chains may not be supported, even if the instruction counts fit within the exported limits. A texture dependency occurs when a texture instruction depends on the result of a previous instruction (ALU or texture) for use as its texture coordinate.

A texture indirection can be considered a node in the texture dependency chain. Each node contains a set of texture instructions which execute in parallel, followed by a sequence of ALU instructions. A dependent texture instruction is one that uses a temporary as an input coordinate rather than an attribute or a parameter. A program with no dependent texture instructions (or no texture instructions at all) will have a single node in its texture dependency chain, and thus a single indirection.

API-level texture indirections are counted by keeping track of which temporaries are read and written within the current node in the texture dependency chain. When a texture instruction is encountered, an indirection may be added and a new node started if either of the following two conditions is true:

1. the source coordinate of the texture instruction is a temporary that has already been written in the current node, either by a previous texture instruction or ALU instruction;

2. the result of the texture instruction is a temporary that has already been read or written in the current node by an ALU instruction.

The texture instruction provoking a new indirection and all subsequent instructions are added to the new node. This process is repeated until the end of the program is encountered. Below is some pseudo-code to describe this:

```
indirections = 1;
tempsOutput = 0;
aluTemps = 0;
while (i = getInst())
{
  if (i.type == TEX)
  {
    if (((i.input.type == TEMP) &&
          (tempsOutput & (1 << i.input.index))) ||
        ((i.op != KILL) && (i.output.type == TEMP) &&
          (aluTemps & (1 << i.output.index))))
    {
      indirections++;
      tempsOutput = 0;
      aluTemps = 0;
    }
  } else {
    if (i.input1.type == TEMP)
      aluTemps |= (1 << i.input1.index);
    if (i.input2 && i.input2.type == TEMP)
      aluTemps |= (1 << i.input2.index);
    if (i.input3 && i.input3.type == TEMP)
      aluTemps |= (1 << i.input3.index);
    if (i.output.type == TEMP)
      aluTemps |= (1 << i.output.index);
  }
  if ((i.op != KILL) && (i.output.type == TEMP))
    tempsOutput |= (1 << i.output.index);
}
```

    For example, the following programs would have 1, 2, and 3
    texture indirections, respectively:

```
      !!ARBfp1.0
      # No texture instructions, but always 1 indirection
      MOV result.color, fragment.color;
      END

      !!ARBfp1.0
      # A simple dependent texture instruction, 2 indirections
      TEMP myColor;
      MUL myColor, fragment.texcoord[0], fragment.texcoord[1];
      TEX result.color, myColor, texture[0], 2D;
      END

      !!ARBfp1.0
      # A more complex example with 3 indirections
      TEMP myColor1, myColor2;
      TEX myColor1, fragment.texcoord[0], texture[0], 2D;
      MUL myColor1, myColor1, myColor1;
      TEX myColor2, fragment.texcoord[1], texture[1], 2D;
      # so far we still only have 1 indirection
      TEX myColor2, myColor1, texture[2], 2D;      # This is #2
      TEX result.color, myColor2, texture[3], 2D;  # And #3
      END
```

Note that writemasks for the temporaries written and swizzles for the temporaries read are not taken into consideration when counting indirections. This makes hand-counting of indirections by a developer an easier task.

Native texture indirections may be counted differently by an implementation to reflect its exact restrictions, to reflect the true dependencies taking into account writemasks and swizzles, and to reflect optimizations such as instruction reordering.

For implementations with no restrictions on the number of indirections, the maximum indirection count will equal the maximum texture instruction count.

(25) How can a program reduce SCS's scalar operand to the fundamental period [-PI,PI]?

RESOLVED: Unlike the individual SIN and COS instructions, SCS requires that its argument be reduced ahead of time to the fundamental period. The reason SCS doesn't perform this operation automatically is that it may make unnecessary redundant work for programs that already have their operand in the correct range. Other programs that do need to reduce their operand simply need to add a block of code before the SCS instruction:

```
     PARAM myParams = { 0.5, -3.14159, 6.28319, 0.15915 };
     MAD myOperand.x, myOperand.x, myParams.w, myParams.x; # a = (a/(2*PI))+0.5
     FRC myOperand.x, myOperand.x;                         # a = frac(a)
     MAD myOperand.x, myOperand.x, myParams.z, myParams.y  # a = (a*2*PI)-PI
     ...
     SCS myResult, myOperand.x;
```

(26) Is depth output from a fragment program guaranteed to be invariant with respect to depth produced via conventional rasterization?

RESOLVED: No. The floating-point representation of depth values output from a fragment program may lead to the output of depth with less precision than the depth output by convention GL rasterization. For example, a floating-point representation with 16 bits of mantissa will certainly produce depth with lesser precision than that of conventional rasterization used in conjunction with a 24-bit depth buffer, where all values are maintained as integers. Be aware of this when mixing conventional GL rendering with fragment program rendering.

(27) How can conventional GL fog application be achieved within a fragment program?

RESOLVED: Program options have been introduced that allow a program to request fog to be applied to the final clamped fragment color before being passed along to the antialiasing application stage. This makes it easy for:

1. developers to request conventional fog behavior
2. implementations with dedicated fog hardware to use it
3. implementations without dedicated fog hardware, so they need not track fog state after compilation, and constantly recompile when fog state changes.

The three mandatory options are ARB_fog_exp, ARB_fog_exp2, and ARB_fog_linear. As these options are mutually exclusive by nature, specifying more than one is not useful. If more than one is specified, the last one encountered in the <optionSequence> will be the one to actually modify the execution environment.

(28) Why have all of the enums, entrypoints, GLX protocol, and spec language shared with ARB_vertex_program been reproduced here?

RESOLVED: The two extensions are independent of one another, in so far as an implementation need not support both of them in order to support one of them. Everything needed to implement or make use of ARB_fragment_program is present in this spec without the need to refer to the ARB_vertex_program spec. When and if these two extensions are incorporated into the core OpenGL, the significant overlap of the two will be collapsed into a single instance of the shared parts.

(29) How might an implementation implement the fog options? To What does the extra resource consumption described in 3.11.4.5.1 correspond?

RESOLVED: The following code snippets reflect possible implementations of the fog options. While an implementation may use other instruction sequences to achieve the same result, or may use external fog hardware if available, all implementations must enforce the API-level resource consumption as described: 2 params, 1 temp, 1 attribute, and 3, 4, or 2 instructions. "finalColor" in the examples below is the color that would otherwise be

```
    "result.color", with components clamped to the range [0,1].
    "result.color.a" is assumed to have already been written, as fog
    blending does not affect the alpha component.

    EXP:
      # Exponential fog
      # f = exp(-d*z)
      #
      PARAM p = {DENSITY/LN(2), NOT USED, NOT USED, NOT USED};
      PARAM fogColor = state.fog.color;
      TEMP fogFactor;
      ATTRIB fogCoord = fragment.fogcoord.x;
      MUL fogFactor.x, p.x, fogCoord.x;
      EX2_SAT fogFactor.x, -fogFactor.x;
      LRP result.color.rgb, fogFactor.x, finalColor, fogColor;

    EXP2:
      #
      # 2nd-order Exponential fog
      # f = exp(-(d*z)^2)
      #
      PARAM p = {DENSITY/SQRT(LN(2)), NOT USED, NOT USED, NOT USED};
      PARAM fogColor = state.fog.color;
      TEMP fogFactor;
      ATTRIB fogCoord = fragment.fogcoord.x;
      MUL fogFactor.x, p.x, fogCoord.x;
      MUL fogFactor.x, fogFactor.x, fogFactor.x;
      EX2_SAT fogFactor.x, -fogFactor.x;
      LRP result.color.rgb, fogFactor.x, finalColor, fogColor;

    LINEAR:
      #
      # Linear fog
      # f = (end-z)/(end-start)
      #
      PARAM p = {-1/(END-START), END/(END-START), NOT USED, NOT USED};
      PARAM fogColor = state.fog.color;
      TEMP fogFactor;
      ATTRIB fogCoord = fragment.fogcoord.x;
      MAD_SAT fogFactor.x, p.x, fogCoord.x, p.y;
      LRP result.color.rgb, fogFactor.x, finalColor, fogColor;
```

(30) Why is the order of operands for the CMP instruction different than the order used by another popular graphics API?

RESOLVED: No other graphics API was used as a basis for the design of ARB_fragment_program except ARB_vertex_program, which did not have a CMP instruction. This independent evolution naturally led to differences in minor details such as order of operands. This discrepancy is noted here to help developers familiar with the other API to avoid this potential pitfall.

(31) Is depth offset applied to the window z value before it enters the fragment program?

RESOLVED: As in the base OpenGL specification, the depth offset generated by polygon offset is added during polygon rasterization.

    The depth value provided to shaders in the fragment.position.z
    attribute already includes polygon offset, if enabled. If the
    depth value is replaced by a fragment program, the polygon offset
    value will NOT be recomputed and added back after fragment program
    execution.

    NOTE: This is probably not desirable for fragment programs that
    modify depth values since the partials used to generate the offset
    may not match the partials of the computed depth value.

New Procedures and Functions

```
    void ProgramStringARB(enum target, enum format, sizei len,
                          const void *string);

    void BindProgramARB(enum target, uint program);

    void DeleteProgramsARB(sizei n, const uint *programs);

    void GenProgramsARB(sizei n, uint *programs);

    void ProgramEnvParameter4dARB(enum target, uint index,
                                  double x, double y, double z, double w);
    void ProgramEnvParameter4dvARB(enum target, uint index,
                                   const double *params);
    void ProgramEnvParameter4fARB(enum target, uint index,
                                  float x, float y, float z, float w);
    void ProgramEnvParameter4fvARB(enum target, uint index,
                                   const float *params);

    void ProgramLocalParameter4dARB(enum target, uint index,
                                    double x, double y, double z, double w);
    void ProgramLocalParameter4dvARB(enum target, uint index,
                                     const double *params);
    void ProgramLocalParameter4fARB(enum target, uint index,
                                    float x, float y, float z, float w);
    void ProgramLocalParameter4fvARB(enum target, uint index,
                                     const float *params);

    void GetProgramEnvParameterdvARB(enum target, uint index,
                                     double *params);
    void GetProgramEnvParameterfvARB(enum target, uint index,
                                     float *params);

    void GetProgramLocalParameterdvARB(enum target, uint index,
                                       double *params);
    void GetProgramLocalParameterfvARB(enum target, uint index,
                                       float *params);

    void GetProgramivARB(enum target, enum pname, int *params);

    void GetProgramStringARB(enum target, enum pname, void *string);

    boolean IsProgramARB(uint program);
```

New Tokens

```
    Accepted by the <cap> parameter of Disable, Enable, and IsEnabled,
    by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv,
    and GetDoublev, and by the <target> parameter of ProgramStringARB,
    BindProgramARB, ProgramEnvParameter4[df][v]ARB,
    ProgramLocalParameter4[df][v]ARB, GetProgramEnvParameter[df]vARB,
    GetProgramLocalParameter[df]vARB, GetProgramivARB and
    GetProgramStringARB.

        FRAGMENT_PROGRAM_ARB                            0x8804

    Accepted by the <format> parameter of ProgramStringARB:

        PROGRAM_FORMAT_ASCII_ARB                        0x8875

    Accepted by the <pname> parameter of GetProgramivARB:

        PROGRAM_LENGTH_ARB                              0x8627
        PROGRAM_FORMAT_ARB                              0x8876
        PROGRAM_BINDING_ARB                             0x8677
        PROGRAM_INSTRUCTIONS_ARB                        0x88A0
        MAX_PROGRAM_INSTRUCTIONS_ARB                    0x88A1
        PROGRAM_NATIVE_INSTRUCTIONS_ARB                 0x88A2
        MAX_PROGRAM_NATIVE_INSTRUCTIONS_ARB             0x88A3
        PROGRAM_TEMPORARIES_ARB                         0x88A4
        MAX_PROGRAM_TEMPORARIES_ARB                     0x88A5
        PROGRAM_NATIVE_TEMPORARIES_ARB                  0x88A6
        MAX_PROGRAM_NATIVE_TEMPORARIES_ARB              0x88A7
        PROGRAM_PARAMETERS_ARB                          0x88A8
        MAX_PROGRAM_PARAMETERS_ARB                      0x88A9
        PROGRAM_NATIVE_PARAMETERS_ARB                   0x88AA
        MAX_PROGRAM_NATIVE_PARAMETERS_ARB               0x88AB
        PROGRAM_ATTRIBS_ARB                             0x88AC
        MAX_PROGRAM_ATTRIBS_ARB                         0x88AD
        PROGRAM_NATIVE_ATTRIBS_ARB                      0x88AE
        MAX_PROGRAM_NATIVE_ATTRIBS_ARB                  0x88AF
        MAX_PROGRAM_LOCAL_PARAMETERS_ARB                0x88B4
        MAX_PROGRAM_ENV_PARAMETERS_ARB                  0x88B5
        PROGRAM_UNDER_NATIVE_LIMITS_ARB                 0x88B6
        PROGRAM_ALU_INSTRUCTIONS_ARB                    0x8805
        PROGRAM_TEX_INSTRUCTIONS_ARB                    0x8806
        PROGRAM_TEX_INDIRECTIONS_ARB                    0x8807
        PROGRAM_NATIVE_ALU_INSTRUCTIONS_ARB             0x8808
        PROGRAM_NATIVE_TEX_INSTRUCTIONS_ARB             0x8809
        PROGRAM_NATIVE_TEX_INDIRECTIONS_ARB             0x880A
        MAX_PROGRAM_ALU_INSTRUCTIONS_ARB                0x880B
        MAX_PROGRAM_TEX_INSTRUCTIONS_ARB                0x880C
        MAX_PROGRAM_TEX_INDIRECTIONS_ARB                0x880D
        MAX_PROGRAM_NATIVE_ALU_INSTRUCTIONS_ARB         0x880E
        MAX_PROGRAM_NATIVE_TEX_INSTRUCTIONS_ARB         0x880F
        MAX_PROGRAM_NATIVE_TEX_INDIRECTIONS_ARB         0x8810

    Accepted by the <pname> parameter of GetProgramStringARB:

        PROGRAM_STRING_ARB                              0x8628
```

Accepted by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

PROGRAM_ERROR_POSITION_ARB	0x864B
CURRENT_MATRIX_ARB	0x8641
TRANSPOSE_CURRENT_MATRIX_ARB	0x88B7
CURRENT_MATRIX_STACK_DEPTH_ARB	0x8640
MAX_PROGRAM_MATRICES_ARB	0x862F
MAX_PROGRAM_MATRIX_STACK_DEPTH_ARB	0x862E
MAX_TEXTURE_COORDS_ARB	0x8871
MAX_TEXTURE_IMAGE_UNITS_ARB	0x8872

Accepted by the <name> parameter of GetString:

PROGRAM_ERROR_STRING_ARB	0x8874

Accepted by the <mode> parameter of MatrixMode:

MATRIX0_ARB	0x88C0
MATRIX1_ARB	0x88C1
MATRIX2_ARB	0x88C2
MATRIX3_ARB	0x88C3
MATRIX4_ARB	0x88C4
MATRIX5_ARB	0x88C5
MATRIX6_ARB	0x88C6
MATRIX7_ARB	0x88C7
MATRIX8_ARB	0x88C8
MATRIX9_ARB	0x88C9
MATRIX10_ARB	0x88CA
MATRIX11_ARB	0x88CB
MATRIX12_ARB	0x88CC
MATRIX13_ARB	0x88CD
MATRIX14_ARB	0x88CE
MATRIX15_ARB	0x88CF
MATRIX16_ARB	0x88D0
MATRIX17_ARB	0x88D1
MATRIX18_ARB	0x88D2
MATRIX19_ARB	0x88D3
MATRIX20_ARB	0x88D4
MATRIX21_ARB	0x88D5
MATRIX22_ARB	0x88D6
MATRIX23_ARB	0x88D7
MATRIX24_ARB	0x88D8
MATRIX25_ARB	0x88D9
MATRIX26_ARB	0x88DA
MATRIX27_ARB	0x88DB
MATRIX28_ARB	0x88DC
MATRIX29_ARB	0x88DD
MATRIX30_ARB	0x88DE
MATRIX31_ARB	0x88DF

Additions to Chapter 2 of the OpenGL 1.3 Specification (OpenGL Operation)

Modify Section 2.1.1, Floating-Point Computation (p. 6)

(modify first paragraph, p. 6) ... The maximum representable magnitude of a floating-point number used to represent position, normal, or texture coordinates must be at least 2^32; the maximum representable magnitude for colors must be at least 2^10. ...

Modify Section 2.7, Vertex Specification (p. 19)

(modify second paragraph, p. 20) Implementations support more than one set of texture coordinates. The commands

```
  void MultiTexCoord{1234}{sifd}(enum texture, T coords);
  void MultiTexCoord{1234}{sifd}v(enum texture, T coords);
```

take the coordinate set to be modified as the <texture> parameter. <texture> is a symbolic constant of the form TEXTUREi, indicating that texture coordinate set i is to be modified. The constants obey TEXTUREi = TEXTURE0 + i (i is in the range 0 to k-1, where k is the implementation-dependent number of texture units defined by MAX_TEXTURE_COORDS_ARB).

Modify Section 2.8, Vertex Arrays (p. 21)

(modify first paragraph, p. 21) ... The client may specify up to 5 plus the value of MAX_TEXTURE_COORDS_ARB arrays: one each to store vertex coordinates...

(modify first paragraph, p. 23) The command

```
  void ClientActiveTexture(enum texture);
```

is used to select the vertex array client state parameters to be modified by the TexCoordPointer command and the array affected by EnableClientState and DisableClientState with parameter TEXTURE_COORD_ARRAY. This command sets the client state variable CLIENT_ACTIVE_TEXTURE. Each texture coordinate set has a client state vector which is selected when this command is invoked. This state vector includes the vertex array state. This call also selects the texture coordinate set state used for queries of client state.

(modify first paragraph, p. 28) If the number of supported texture coordinate sets (the value of MAX_TEXTURE_COORDS_ARB) is k, ...

Modify Section 2.10.2, Matrices (p. 31)

(modify first paragraph, p. 31) The projection matrix and model-view matrix are set and modified with a variety of commands. The affected matrix is determined by the current matrix mode. The current matrix mode is set with

```
  void MatrixMode(enum mode);
```

which takes one of the pre-defined constants TEXTURE, MODELVIEW, COLOR, PROJECTION, or MATRIX<i>_ARB as the argument. In the case of MATRIX<i>_ARB, <i> is an integer between 0 and <n>-1 indicating one of <n> program matrices where <n> is the value of the implementation defined constant MAX_PROGRAM_MATRICES_ARB. Such program matrices are described in section 3.11.7. TEXTURE is described later in section 2.10.2, and COLOR is described in section 3.6.3. If the current matrix mode is MODELVIEW, then matrix operations apply to the model-view matrix; if PROJECTION, then they apply to the projection matrix.

(modify first paragraph, p. 34) For each texture coordinate set, a 4x4 matrix is applied to the corresponding texture coordinates...

(modify first and second paragraphs, p. 35) The command

```
  void ActiveTexture(enum texture);
```

specifies the active texture unit selector, ACTIVE_TEXTURE. Each texture unit contains up to two distinct sub-units: a texture coordinate processing unit (consisting of a texture matrix stack and texture coordinate generation state) and a texture image unit (consisting of all the texture state defined in Section 3.8). In implementations with a different number of supported texture coordinate sets and texture image units, some texture units may consist of only one of the two sub-units.

The active texture unit selector specifies the texture coordinate set accessed by commands involving texture coordinate processing. Such commands include those accessing the current matrix stack (if MATRIX_MODE is TEXTURE), TexGen (section 2.10.4), Enable/Disable (if any texture coordinate generation enum is selected), as well as queries of the current texture coordinates and current raster texture coordinates. If the texture coordinate set number corresponding to the current value of ACTIVE_TEXTURE is greater than or equal to the implementation-dependent constant MAX_TEXTURE_COORDS_ARB, the error INVALID_OPERATION is generated by any such command.

The active texture unit selector also selects the texture image unit accessed by commands involving texture image processing (section 3.8). Such commands include all variants of TexEnv, TexParameter, and TexImage commands, BindTexture, Enable/Disable for any texture target (e.g., TEXTURE_2D), and queries of all such state. If the texture image unit number corresponding to the current value of ACTIVE_TEXTURE is greater than or equal to the implementation-dependent constant MAX_TEXTURE_IMAGE_UNITS_ARB, the error INVALID_OPERATION is generated by any such command.

ActiveTexture generates the error INVALID_ENUM if an invalid <texture> is specified. <texture> is a symbolic constant of the form TEXTUREi, indicating that texture unit i is to be modified. The constants obey TEXTUREi = TEXTURE0 + i (i is in the range 0 to k-1, where k is the larger of the MAX_TEXTURE_COORDS_ARB and MAX_TEXTURE_IMAGE_UNITS_ARB). For compatibility with old OpenGL specifications, the implementation-dependent constant MAX_TEXTURE_UNITS specifies the number of conventional texture units supported by the implementation. Its value must be no larger than the minimum of MAX_TEXTURE_COORDS_ARB and MAX_TEXTURE_IMAGE_UNITS_ARB.

(modify last paragraph, p. 35) The state required to implement transformations consists of a <n>-value integer indicating the current matrix mode (where <n> is 4 + the number of supported texture and program matrices), a stack of at least two 4x4 matrices for each of COLOR, PROJECTION, and TEXTURE with associated stack pointers, <n> stacks (where <n> is at least 8) of at least one 4x4 matrix for each MATRIX<i>_ARB with associated stack pointers, and a stack of at least 32 4x4 matrices with an associated stack pointer for MODELVIEW. Initially, there is only one matrix on each stack, and all matrices are set to the identity. The initial matrix mode is MODELVIEW. The initial value of ACTIVE_TEXTURE is TEXTURE0.

Additions to Chapter 3 of the OpenGL 1.3 Specification (Rasterization)

Modify Chapter 3, Introduction (p. 58)

(modify first paragraph, p. 58) ... Figure 3.1 diagrams the rasterization process. The color value assigned to a fragment is initially determined by the rasterization operations (sections 3.3 through 3.7) and modified by either the execution of the texturing, color sum, and fog operations as defined in sections 3.8, 3.9, and 3.10, or of a fragment program defined in section 3.11. The final depth value is initially determined by the rasterization operations and may be modified or replaced by a fragment program.

(modify Figure 3.1)

```
             _ +---------------+   FRAGMENT_PROGRAM_ARB
             /||    Point      |          enable
            /  | Rasterization |\           |
           /   +---------------+ \          V  o-------------+
   From   /    +---------------+  \                          |
Primitive ---> |     Line      |---+++--->o    o             |
 Assembly \    | Rasterization |  / ||         |             |
           \   +---------------+ /  ||         |             |
            \  +---------------+/   ||   +-----+-----+  +----+-----+
             \||    Polygon    |    ||   | Texturing |  | Fragment |
             - | Rasterization |   / |   +-----+-----+  | Program  |
               +---------------+  /  |         |        +----+-----+
               +---------------+ /   |   +-----+-----+       |
               |     Pixel     |/    |   | Color Sum |       |
DrawPixels --> |   Rectangle   |    /    +-----+-----+       |
               | Rasterization |   /           |             V
               +---------------+  /      +-----+-----+
               +---------------+ /       |    Fog    |---> Fragments
  Bitmap ----> |    Bitmap     |/        +-----------+
               | Rasterization |
               +---------------+
```

Modify Section 3.3, Points (p. 63)

(modify first and second paragraphs, p. 64) All fragments produced in rasterizing a non-antialiased point are assigned the same associated data, which are those of the vertex corresponding to the point. (delete reference to divide by q)

If antialiasing is enabled, then ... The data associated with each fragment are otherwise the data associated with the point being rasterized. (delete reference to divide by q)

Modify Section 3.4.1, Basic Line Segment Rasterization (p. 66)

(modify first paragraph, p. 68) ... (Note that t=0 at p_a and t=1 at p_b). The value of an associated datum f from the fragment center, whether it be R, G, B, or A (in RGBA mode) or a color index (in color index mode) or the s, t, r, or q texture coordinate or the clip w coordinate (the depth value, window z, must be found using equation 3.3, below), is found as

$$f = \frac{(1-t)*(f_a/w_a) + t*(f_b/w_b)}{(1-t)*(1/w_a) + t*(1/w_b)} \qquad (3.2)$$

where f_a and f_b are the data associated with the starting and ending endpoints of the segment, respectively; w_a and w_b are the clip w coordinates of the starting and ending endpoints of the segments, respectively. Note that linear interpolation would use

$$f = (1-t)*f_a + t*f_b. \qquad (3.3)$$

... A GL implementation may choose to approximate equation 3.2 with 3.3, but this will normally lead to inacceptable distortion effects when interpolating texture coordinates or clip w coordinates.

Modify Section 3.5.1, Basic Polygon Rasterization (p. 73)

(modify third and fourth paragraphs, p. 74) Denote a datum at p_a, p_b, or p_c as f_a, f_b, or f_c, respectively. Then the value f of a datum at a fragment produced by rasterizing a triangle is given by

$$f = \frac{a*(f_a/w_a) + b*(f_b/w_b) + c*(f_c/w_c)}{a*(1/w_a) + b*(1/w_b) + c*(1/w_c)} \qquad (3.4)$$

where w_a, w_b, and w_c are the clip w coordinates of p_a, p_b, and p_c, respectively. a, b, and c are the barycentric coordinates of the fragment for which the data are produced. a, b, and c must correspond precisely to the ... at the fragment's center.

Just as with line segment rasterization, equation 3.4 may be approximated by

$$f = a*f_a + b*f_b + c*f_c;$$

this may yield ... for texture coordinates or clip w coordinates.

Modify Section 3.6.4, Rasterization of Pixel Rectangles (p. 91)

(modify third paragraph, p. 103) A fragment arising from a group ... the color and texture coordinates are given by those associated with the current raster position. (delete reference to divide by q) Groups arising from DrawPixels...

Modify Section 3.7, Bitmaps (p. 113)

(modify third paragraph, p. 114) Otherwise, a rectangular array ... The associated data for each fragment are those associated with the current raster position. (delete reference to divide by q) Once the fragments have been produced ...

Modify Section 3.8, Texturing (p. 115)

(add new paragraphs before first paragraph, p. 115) Texture coordinate sets are mapped to RGBA colors for application to primitives in one of two modes. The first mode, described in this and subsequent sections, is GL's conventional multitexture pipeline, describing texture environment and texture application. The second mode, referred to as fragment program mode and described in section 3.11, applies textures, color sum, and fog as specified in an application-supplied fragment program.

The fragment program mode is enabled and disabled using the generic Enable and Disable commands, respectively, with the symbolic constant FRAGMENT_PROGRAM_ARB. The required state is one bit indicating whether the fragment program mode is enabled or disabled. In the initial state, the fragment program mode is disabled. When fragment program mode is enabled, texturing, color sum, and fog application stages are ignored and a general purpose program is executed instead.

(modify first and second paragraph, p. 115) Conventional texturing is employed when fragment program mode is disabled. Texturing maps ... color of an image at the location indicated by a fragment's texture coordinates to modify the fragment's primary RGBA color. Texturing does not affect the secondary color.

An implementation may support texturing using more than one image at a time. In this case the fragment carries multiple sets of texture coordinates which are used to index ...

(add paragraph before 1st paragraph, p. 116) Except when in fragment program mode (section 3.11), the (s,t,r) texture coordinates used for texturing are the values s/q, t/q, and r/q, respectively, where s, t, r, and q are the texture coordinates associated with the fragment. When in fragment program mode, the (s,t,r) texture coordinates are specified by the program. If q is less than or equal to zero, the results of texturing are undefined.

Modify Section 3.8.7, Texture Minification (p. 135)

(add new paragraph after first paragraph, p. 137) When fragment program mode is enabled, the derivatives of the coordinates may be ill-defined or non-existent. As a result, the implementation is free to approximate these derivatives with such techniques as differencing. The only requirement is that texture samples be equivalent across the two modes. In other words, the texture sample chosen for a fragment of a primitive must be invariant between fragment program mode and conventional mode subject to the rules set forth in Appendix A, Invariance.

Modify Section 3.8.13, Texture Application (p. 149)

(modify fourth paragraph, p. 152) Texturing is enabled and disabled individually for each texture unit. If texturing is disabled for one of the units, then the fragment resulting from the previous unit is passed unaltered to the following unit. Individual texture units beyond those specified by MAX_TEXTURE_UNITS may be incomplete and are always treated as disabled.

Insert a new Section 3.11, (p. 154), between existing sections 3.10 and 3.11. Renumber 3.11, Antialiasing Application, to 3.12.

3.11 Fragment Programs

The conventional GL texturing model described in section 3.8 is a configurable but essentially hard-wired sequence of per-fragment computations based on a canonical set of per-fragment parameters and texturing-related state such as texture images, texture parameters, and texture environment parameters. The general success and utility of the conventional GL texturing model reflects its basic correspondence to the typical texturing requirements of 3D applications.

However when the conventional GL texturing model is not sufficient, the fragment program mode provides a substantially more flexible model for generating fragment colors. The fragment program mode permits applications to define their own fragment programs.

A fragment program is a character string that specifies a sequence of operations to perform. Fragment program instructions are typically 4-component vector operations that operate on per-fragment attributes and program parameters. Fragment programs execute on a per-fragment basis and operate on each fragment completely independently from any other fragments. Fragment programs execute a finite fixed sequence of instructions with no branching or looping. Fragment programs execute without data hazards so results computed in one instruction can be used immediately afterwards. The result of a fragment program is a set of fragment result registers that becomes the color used by antialiasing application and/or a depth value used in place of the interpolated depth value generated by conventional rasterization.

In fragment program mode, the color sum is subsumed by the fragment program. An application desiring the primary and secondary colors to be summed must explicitly include this operation in its program.

Fragment programs are defined to operate only in RGBA mode. The results of fragment program execution are undefined if the GL is in color index mode.

3.11.1 Program Objects

The GL provides one or more program targets, each identifying a portion of the GL that can be controlled through application-specified programs. The program target for fragment programs is FRAGMENT_PROGRAM_ARB. Each program target has an associated program object, called the current program object. Each program target also has a default program object, which is initially the current program object.

Each program object has an associated program string. The command

```
  ProgramStringARB(enum target, enum format, sizei len,
                   const void *string);
```

updates the program string for the current program object for <target>. <format> describes the format of the program string, which must currently be PROGRAM_FORMAT_ASCII_ARB. <string> is a pointer to the array of bytes representing the program string being loaded, which need not be null-terminated. The length of the array is given by <len>. If <string> is null-terminated, <len> should not include the terminator.

When a program string is loaded, it is interpreted according to syntactic and semantic rules corresponding to the program target specified by <target>. If a program violates the syntactic or semantic restrictions of the program target, ProgramStringARB generates the error INVALID_OPERATION. An implementation may also generate the error INVALID_OPERATION if the program would exceed the native resource limits defined in section 6.1.12. A program which fails to load due to exceeding native resource limits must always fail, regardless of any other GL state.

Additionally, ProgramString will update the program error position (PROGRAM_ERROR_POSITION_ARB) and error string (PROGRAM_ERROR_STRING_ARB). If a program fails to load, the value of the program error position is set to the ubyte offset into the specified program string indicating where the first program error was detected. If the program fails to load because of a semantic restriction that is not detected until the program is fully scanned, the error position is set to the value of <len>. If a program loads successfully, the error position is set to the value negative one. The implementation-dependent program error string contains one or more error or warning messages. If a program loads succesfully, the error string may either contain warning messages or be empty.

Each program object has an associated array of program local parameters. The number and type of program local parameters is target- and implementation-dependent. For fragment programs, program local parameters are four-component floating-point vectors. The number of vectors is given by the implementation-dependent constant MAX_PROGRAM_LOCAL_PARAMETERS_ARB, which must be at least 24. The commands

```
      void ProgramLocalParameter4fARB(enum target, uint index,
                                      float x, float y, float z, float w);
      void ProgramLocalParameter4fvARB(enum target, uint index,
                                       const float *params);
      void ProgramLocalParameter4dARB(enum target, uint index,
                                      double x, double y, double z, double w);
      void ProgramLocalParameter4dvARB(enum target, uint index,
                                       const double *params);
```

update the values of the program local parameter numbered <index> belonging to the program object currently bound to <target>. For ProgramLocalParameter4fARB and ProgramLocalParameter4dARB, the four components of the parameter are updated with the values of <x>, <y>, <z>, and <w>, respectively. For ProgramLocalParameter4fvARB and ProgramLocalParameter4dvARB, the four components of the parameter are updated with the array of four values pointed to by <params>. The error INVALID_VALUE is generated if <index> is greater than or equal to the number of program local parameters supported by <target>.

Additionally, each program target has an associated array of program environment parameters. Unlike program local parameters, program environment parameters are shared by all program objects of a given target. The number and type of program environment parameters is target- and implementation-dependent. For fragment programs, program environment parameters are four-component floating-point vectors. The number of vectors is given by the implementation-dependent constant MAX_PROGRAM_ENV_PARAMETERS_ARB, which must be at least 24. The commands

```
      void ProgramEnvParameter4fARB(enum target, uint index,
                                    float x, float y, float z, float w);
      void ProgramEnvParameter4fvARB(enum target, uint index,
                                     const float *params);
      void ProgramEnvParameter4dARB(enum target, uint index,
                                    double x, double y, double z, double w);
      void ProgramEnvParameter4dvARB(enum target, uint index,
                                     const double *params);
```

update the values of the program environment parameter numbered <index> for the given program target <target>. For ProgramEnvParameter4fARB and ProgramEnvParameter4dARB, the four components of the parameter are updated with the values of <x>, <y>, <z>, and <w>, respectively. For ProgramEnvParameter4fvARB and ProgramEnvParameter4dvARB, the four components of the parameter are updated with the array of four values pointed to by <params>. The error INVALID_VALUE is generated if <index> is greater than or equal to the number of program environment parameters supported by <target>.

Each program target has a default program object. Additionally, named program objects can be created and operated upon. The name space for program objects is the positive integers and is shared by programs of all targets. The name zero is reserved by the GL.

A named program object is created by binding an unused program object name to a valid program target. The binding is effected by calling

```
  BindProgramARB(enum target, uint program);
```

with <target> set to the desired program target and <program> set to the unused program name. The resulting program object has a program target given by <target> and is assigned target-specific default values (see section 3.11.8 for fragment programs). BindProgramARB may also be used to bind an existing program object to a program target. If <program> is zero, the default program object for <target> is bound. If <program> is the name of an existing program object whose associated program target is <target>, the named program object is bound. The error INVALID_OPERATION is generated if <program> names an existing program object whose associated program target is anything other than <target>.

Programs objects are deleted by calling

```
  void DeleteProgramsARB(sizei n, const uint *programs);
```

<programs> contains <n> names of programs to be deleted. After a program object is deleted, its name is again unused. If a program object that is bound to any target is deleted, it is as though BindProgramARB is first executed with same target and a <program> of zero. Unused names in <programs> are silently ignored, as is the value zero.

The command

```
  void GenProgramsARB(sizei n, uint *programs);
```

returns <n> currently unused program names in <programs>. These names are marked as used, for the purposes of GenProgramsARB only, but objects are created only when they are first bound using BindProgramARB.

3.11.2 Fragment Program Grammar and Semantic Restrictions

Fragment program strings are specified as an array of ASCII characters containing the program text. When a fragment program is loaded by a call to ProgramStringARB, the program string is parsed into a set of tokens possibly separated by whitespace. Spaces, tabs, newlines, carriage returns, and comments are considered whitespace. Comments begin with the character "#" and are terminated by a newline, a carriage return, or the end of the program array.

The Backus-Naur Form (BNF) grammar below specifies the syntactically valid sequences for fragment programs. The set of valid tokens can be inferred from the grammar. The token "" represents an empty string and is used to indicate optional rules. A program is invalid if it contains any undefined tokens or characters.

A fragment program is required to begin with the header string "!!ARBfp1.0", without any preceding whitespace. This string

```
    identifies the subsequent program text as a fragment program
    (version 1.0) that should be parsed according to the following
    grammar and semantic rules.  Program string parsing begins with the
    character immediately following the header string.

    <program>              ::= <optionSequence> <statementSequence> "END"

    <optionSequence>       ::= <optionSequence> <option>
                             | ""

    <option>               ::= "OPTION" <identifier> ";"

    <statementSequence>    ::= <statementSequence> <statement>
                             | ""

    <statement>            ::= <instruction> ";"
                             | <namingStatement> ";"

    <instruction>          ::= <ALUInstruction>
                             | <TexInstruction>

    <ALUInstruction>       ::= <VECTORop_instruction>
                             | <SCALARop_instruction>
                             | <BINSCop_instruction>
                             | <BINop_instruction>
                             | <TRIop_instruction>
                             | <SWZ_instruction>

    <TexInstruction>       ::= <SAMPLE_instruction>
                             | <KIL_instruction>

    <VECTORop_instruction> ::= <VECTORop> <maskedDstReg> ","
                               <vectorSrcReg>

    <VECTORop>             ::= "ABS" | "ABS_SAT"
                             | "FLR" | "FLR_SAT"
                             | "FRC" | "FRC_SAT"
                             | "LIT" | "LIT_SAT"
                             | "MOV" | "MOV_SAT"

    <SCALARop_instruction> ::= <SCALARop> <maskedDstReg> ","
                               <scalarSrcReg>

    <SCALARop>             ::= "COS" | "COS_SAT"
                             | "EX2" | "EX2_SAT"
                             | "LG2" | "LG2_SAT"
                             | "RCP" | "RCP_SAT"
                             | "RSQ" | "RSQ_SAT"
                             | "SIN" | "SIN_SAT"
                             | "SCS" | "SCS_SAT"

    <BINSCop_instruction>  ::= <BINSCop> <maskedDstReg> ","
                               <scalarSrcReg> "," <scalarSrcReg>

    <BINSCop>              ::= "POW" | "POW_SAT"
    <BINop_instruction>    ::= <BINop> <maskedDstReg> ","
                               <vectorSrcReg> "," <vectorSrcReg>

    <BINop>                ::= "ADD" | "ADD_SAT"
                             | "DP3" | "DP3_SAT"
                             | "DP4" | "DP4_SAT"
                             | "DPH" | "DPH_SAT"
                             | "DST" | "DST_SAT"
                             | "MAX" | "MAX_SAT"
                             | "MIN" | "MIN_SAT"
                             | "MUL" | "MUL_SAT"
                             | "SGE" | "SGE_SAT"
                             | "SLT" | "SLT_SAT"
                             | "SUB" | "SUB_SAT"
                             | "XPD" | "XPD_SAT"

    <TRIop_instruction>    ::= <TRIop> <maskedDstReg> ","
                               <vectorSrcReg> "," <vectorSrcReg> ","
                               <vectorSrcReg>

    <TRIop>                ::= "CMP" | "CMP_SAT"
                             | "LRP" | "LRP_SAT"
                             | "MAD" | "MAD_SAT"

    <SWZ_instruction>      ::= <SWZop> <maskedDstReg> ","
                               <srcReg> "," <extendedSwizzle>

    <SWZop>                ::= "SWZ" | "SWZ_SAT"

    <SAMPLE_instruction>   ::= <SAMPLEop> <maskedDstReg> ","
                               <vectorSrcReg> "," <texImageUnit> ","
                               <texTarget>

    <SAMPLEop>             ::= "TEX" | "TEX_SAT"
                             | "TXP" | "TXP_SAT"
                             | "TXB" | "TXB_SAT"

    <KIL_instruction>      ::= "KIL" <vectorSrcReg>

    <texImageUnit>         ::= "texture" <optTexImageUnitNum>

    <texTarget>            ::= "1D"
                             | "2D"
                             | "3D"
                             | "CUBE"
                             | "RECT"

    <optTexImageUnitNum>   ::= ""
                             | "[" <texImageUnitNum> "]"

    <texImageUnitNum>      ::= <integer> from 0 to
                               MAX_TEXTURE_IMAGE_UNITS_ARB-1

    <scalarSrcReg>         ::= <optionalSign> <srcReg> <scalarSuffix>

    <vectorSrcReg>         ::= <optionalSign> <srcReg> <optionalSuffix>
    <maskedDstReg>          ::= <dstReg> <optionalMask>

    <extendedSwizzle>       ::= <xyzwExtendedSwizzle>
                              | <rgbaExtendedSwizzle>

    <xyzwExtendedSwizzle>   ::= <xyzwExtSwizComp> "," <xyzwExtSwizComp> ","
                                <xyzwExtSwizComp> "," <xyzwExtSwizComp>

    <rgbaExtendedSwizzle>   ::= <rgbaExtSwizComp> "," <rgbaExtSwizComp> ","
                                <rgbaExtSwizComp> "," <rgbaExtSwizComp>

    <xyzwExtSwizComp>       ::= <optionalSign> <xyzwExtSwizSel>

    <rgbaExtSwizComp>       ::= <optionalSign> <rgbaExtSwizSel>

    <xyzwExtSwizSel>        ::= "0"
                              | "1"
                              | <xyzwComponent>

    <rgbaExtSwizSel>        ::= "0"
                              | "1"
                              | <rgbaComponent>

    <srcReg>                ::= <fragmentAttribReg>
                              | <temporaryReg>
                              | <progParamReg>

    <dstReg>                ::= <temporaryReg>
                              | <fragmentResultReg>

    <fragmentAttribReg>     ::= <establishedName>
                              | <fragAttribBinding>

    <temporaryReg>          ::= <establishedName>

    <progParamReg>          ::= <progParamSingle>
                              | <progParamArray> "[" <progParamArrayAbs> "]"
                              | <paramSingleItemUse>

    <progParamSingle>       ::= <establishedName>

    <progParamArray>        ::= <establishedName>

    <progParamArrayAbs>     ::= <integer>

    <fragmentResultReg>     ::= <establishedName>
                              | <resultBinding>

    <scalarSuffix>          ::= "." <component>

    <optionalSuffix>        ::= ""
                              | "." <component>
                              | "." <xyzwComponent> <xyzwComponent>
                                    <xyzwComponent> <xyzwComponent>
                              | "." <rgbaComponent> <rgbaComponent>
                                    <rgbaComponent> <rgbaComponent>
    <component>             ::= <xyzwComponent>
                              | <rgbaComponent>

    <xyzwComponent>         ::= "x" | "y" | "z" | "w"

    <rgbaComponent>         ::= "r" | "g" | "b" | "a"

    <optionalMask>          ::= ""
                              | <xyzwMask>
                              | <rgbaMask>

    <xyzwMask>              ::= "." "x"
                              | "." "y"
                              | "." "xy"
                              | "." "z"
                              | "." "xz"
                              | "." "yz"
                              | "." "xyz"
                              | "." "w"
                              | "." "xw"
                              | "." "yw"
                              | "." "xyw"
                              | "." "zw"
                              | "." "xzw"
                              | "." "yzw"
                              | "." "xyzw"

    <rgbaMask>              ::= "." "r"
                              | "." "g"
                              | "." "rg"
                              | "." "b"
                              | "." "rb"
                              | "." "gb"
                              | "." "rgb"
                              | "." "a"
                              | "." "ra"
                              | "." "ga"
                              | "." "rga"
                              | "." "ba"
                              | "." "rba"
                              | "." "gba"
                              | "." "rgba"

    <namingStatement>       ::= <ATTRIB_statement>
                              | <PARAM_statement>
                              | <TEMP_statement>
                              | <OUTPUT_statement>
                              | <ALIAS_statement>

    <ATTRIB_statement>      ::= "ATTRIB" <establishName> "="
                                  <fragAttribBinding>

    <fragAttribBinding>     ::= "fragment" "." <fragAttribItem>
    <fragAttribItem>       ::= "color" <optColorType>
                             | "texcoord" <optTexCoordNum>
                             | "fogcoord"
                             | "position"

    <PARAM_statement>      ::= <PARAM_singleStmt>
                             | <PARAM_multipleStmt>

    <PARAM_singleStmt>     ::= "PARAM" <establishName> <paramSingleInit>

    <PARAM_multipleStmt>   ::= "PARAM" <establishName> "[" <optArraySize> "]"
                                   <paramMultipleInit>

    <optArraySize>         ::= ""
                             | <integer> from 1 to MAX_PROGRAM_PARAMETERS_ARB
                                 (maximum number of allowed program
                                  parameter bindings)

    <paramSingleInit>      ::= "=" <paramSingleItemDecl>

    <paramMultipleInit>    ::= "=" "{" <paramMultInitList> "}"

    <paramMultInitList>    ::= <paramMultipleItem>
                             | <paramMultipleItem> "," <paramMultInitList>

    <paramSingleItemDecl>  ::= <stateSingleItem>
                             | <programSingleItem>
                             | <paramConstDecl>

    <paramSingleItemUse>   ::= <stateSingleItem>
                             | <programSingleItem>
                             | <paramConstUse>

    <paramMultipleItem>    ::= <stateMultipleItem>
                             | <programMultipleItem>
                             | <paramConstDecl>

    <stateMultipleItem>    ::= <stateSingleItem>
                             | "state" "." <stateMatrixRows>

    <stateSingleItem>      ::= "state" "." <stateMaterialItem>
                             | "state" "." <stateLightItem>
                             | "state" "." <stateLightModelItem>
                             | "state" "." <stateLightProdItem>
                             | "state" "." <stateTexEnvItem>
                             | "state" "." <stateFogItem>
                             | "state" "." <stateDepthItem>
                             | "state" "." <stateMatrixRow>

    <stateMaterialItem>    ::= "material" <optFaceType> "." <stateMatProperty>

    <stateMatProperty>     ::= "ambient"
                             | "diffuse"
                             | "specular"
                             | "emission"
                             | "shininess"
    <stateLightItem>       ::= "light" "[" <stateLightNumber> "]" "."
                                 <stateLightProperty>

    <stateLightProperty>   ::= "ambient"
                             | "diffuse"
                             | "specular"
                             | "position"
                             | "attenuation"
                             | "spot" "." <stateSpotProperty>
                             | "half"

    <stateSpotProperty>    ::= "direction"

    <stateLightModelItem>  ::= "lightmodel" <stateLModProperty>

    <stateLModProperty>    ::= "." "ambient"
                             | <optFaceType> "." "scenecolor"

    <stateLightProdItem>   ::= "lightprod" "[" <stateLightNumber> "]"
                                 <optFaceType> "." <stateLProdProperty>

    <stateLProdProperty>   ::= "ambient"
                             | "diffuse"
                             | "specular"

    <stateLightNumber>     ::= <integer> from 0 to MAX_LIGHTS-1

    <stateTexEnvItem>      ::= "texenv" <optLegacyTexUnitNum> "."
                                 <stateTexEnvProperty>

    <stateTexEnvProperty>  ::= "color"

    <optLegacyTexUnitNum>  ::= ""
                             | "[" <legacyTexUnitNum> "]"

    <legacyTexUnitNum>     ::= <integer> from 0 to MAX_TEXTURE_UNITS-1

    <stateFogItem>         ::= "fog" "." <stateFogProperty>

    <stateFogProperty>     ::= "color"
                             | "params"

    <stateDepthItem>       ::= "depth" "." <stateDepthProperty>

    <stateDepthProperty>   ::= "range"

    <stateMatrixRow>       ::= <stateMatrixItem> "." "row" "["
                                  <stateMatrixRowNum> "]"

    <stateMatrixRows>      ::= <stateMatrixItem> <optMatrixRows>

    <optMatrixRows>        ::= ""
                             | "." "row" "[" <stateMatrixRowNum> ".."
                                  <stateMatrixRowNum> "]"

    <stateMatrixItem>      ::= "matrix" "." <stateMatrixName>
                               <stateOptMatModifier>
    <stateOptMatModifier>  ::= ""
                             | "." <stateMatModifier>

    <stateMatModifier>     ::= "inverse"
                             | "transpose"
                             | "invtrans"

    <stateMatrixRowNum>    ::= <integer> from 0 to 3

    <stateMatrixName>      ::= "modelview" <stateOptModMatNum>
                             | "projection"
                             | "mvp"
                             | "texture" <optTexCoordNum>
                             | "palette" "[" <statePaletteMatNum> "]"
                             | "program" "[" <stateProgramMatNum> "]"

    <stateOptModMatNum>    ::= ""
                             | "[" <stateModMatNum> "]"

    <stateModMatNum>       ::= <integer> from 0 to MAX_VERTEX_UNITS_ARB-1

    <optTexCoordNum>       ::= ""
                             | "[" <texCoordNum> "]"

    <texCoordNum>          ::= <integer> from 0 to MAX_TEXTURE_COORDS_ARB-1

    <statePaletteMatNum>   ::= <integer> from 0 to MAX_PALETTE_MATRICES_ARB-1

    <stateProgramMatNum>   ::= <integer> from 0 to MAX_PROGRAM_MATRICES_ARB-1

    <programSingleItem>    ::= <progEnvParam>
                             | <progLocalParam>

    <programMultipleItem>  ::= <progEnvParams>
                             | <progLocalParams>

    <progEnvParams>        ::= "program" "." "env"
                                 "[" <progEnvParamNums> "]"

    <progEnvParamNums>     ::= <progEnvParamNum>
                             | <progEnvParamNum> ".." <progEnvParamNum>

    <progEnvParam>         ::= "program" "." "env"
                                 "[" <progEnvParamNum> "]"

    <progLocalParams>      ::= "program" "." "local"
                                 "[" <progLocalParamNums> "]"

    <progLocalParamNums>   ::= <progLocalParamNum>
                             | <progLocalParamNum> ".." <progLocalParamNum>

    <progLocalParam>       ::= "program" "." "local"
                                 "[" <progLocalParamNum> "]"

    <progEnvParamNum>      ::= <integer> from 0 to
                               MAX_PROGRAM_ENV_PARAMETERS_ARB - 1
    <progLocalParamNum>    ::= <integer> from 0 to
                               MAX_PROGRAM_LOCAL_PARAMETERS_ARB - 1

    <paramConstDecl>       ::= <paramConstScalarDecl>
                             | <paramConstVector>

    <paramConstUse>        ::= <paramConstScalarUse>
                             | <paramConstVector>

    <paramConstScalarDecl> ::= <signedFloatConstant>

    <paramConstScalarUse>  ::= <floatConstant>

    <paramConstVector>     ::= "{" <signedFloatConstant> "}"
                             | "{" <signedFloatConstant> ","
                                   <signedFloatConstant> "}"
                             | "{" <signedFloatConstant> ","
                                   <signedFloatConstant> ","
                                   <signedFloatConstant> "}"
                             | "{" <signedFloatConstant> ","
                                   <signedFloatConstant> ","
                                   <signedFloatConstant> ","
                                   <signedFloatConstant> "}"

    <signedFloatConstant>  ::= <optionalSign> <floatConstant>

    <floatConstant>        ::= see text

    <optionalSign>         ::= ""
                             | "-"
                             | "+"

    <TEMP_statement>       ::= "TEMP" <varNameList>

    <varNameList>          ::= <establishName>
                             | <establishName> "," <varNameList>

    <OUTPUT_statement>     ::= "OUTPUT" <establishName> "="
                                 <resultBinding>

    <resultBinding>        ::= "result" "." "color"
                             | "result" "." "depth"

    <optFaceType>          ::= ""
                             | "." "front"
                             | "." "back"

    <optColorType>         ::= ""
                             | "." "primary"
                             | "." "secondary"

    <ALIAS_statement>      ::= "ALIAS" <establishName> "="
                                 <establishedName>

    <establishName>        ::= <identifier>
```

```
<establishedName>      ::= <identifier>

<identifier>           ::= see text
```

The <integer> rule matches an integer constant. The integer consists of a sequence of one or more digits ("0" through "9").

The <floatConstant> rule matches a floating-point constant consisting of an integer part, a decimal point, a fraction part, an "e" or "E", and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of one or more digits ("0" through "9"). Either the integer part or the fraction parts (not both) may be missing; either the decimal point or the "e" (or "E") and the exponent (not both) may be missing.

The <identifier> rule matches a sequence of one or more letters ("A" through "Z", "a" through "z"), digits ("0" through "9), underscores ("_"), or dollar signs ("$"); the first character must not be a number. Upper and lower case letters are considered different (names are case-sensitive). The following strings are reserved keywords and may not be used as identifiers:

```
ABS, ABS_SAT, ADD, ADD_SAT, ALIAS, ATTRIB, CMP, CMP_SAT, COS,
COS_SAT, DP3, DP3_SAT, DP4, DP4_SAT, DPH, DPH_SAT, DST, DST_SAT,
END, EX2, EX2_SAT, FLR, FLR_SAT, FRC, FRC_SAT, KIL, LG2,
LG2_SAT, LIT, LIT_SAT, LRP, LRP_SAT, MAD, MAD_SAT, MAX, MAX_SAT,
MIN, MIN_SAT, MOV, MOV_SAT, MUL, MUL_SAT, OPTION, OUTPUT, PARAM,
POW, POW_SAT, RCP, RCP_SAT, RSQ, RSQ_SAT, SIN, SIN_SAT, SCS,
SCS_SAT, SGE, SGE_SAT, SLT, SLT_SAT, SUB, SUB_SAT, SWZ, SWZ_SAT,
TEMP, TEX, TEX_SAT, TXB, TXB_SAT, TXP, TXP_SAT, XPD, XPD_SAT,
fragment, program, result, state, and texture.
```

The error INVALID_OPERATION is generated if a fragment program fails to load because it is not syntactically correct or for one of the semantic restrictions described in the following sections.

A successfully loaded fragment program is parsed into a sequence of instructions. Each instruction is identified by its tokenized name. The operation of these instructions when executed is defined in section 3.11.5.

A successfully loaded program string replaces the program string previously loaded into the specified program object. If the OUT_OF_MEMORY error is generated by ProgramStringARB, no change is made to the previous contents of the current program object.

3.11.3 Fragment Program Variables

Fragment programs may access a number of different variables during their execution. The following sections define the variables that can be declared and used by a fragment program.

Explicit variable declarations allow a fragment program to establish a variable name that can be used to refer to a specified resource in subsequent instructions. A fragment program will fail to load if it declares the same variable name more than once or if it refers to a variable name that has not been previously declared in the program string.

Implicit variable declarations allow a fragment program to use the name of certain available resources by name.

3.11.3.1 Fragment Attributes

Fragment program attribute variables are a set of four-component floating-point vectors holding the attributes of the fragment being processed. Fragment attribute variables are read-only during fragment program execution.

Fragment attribute variables can be declared explicitly using the <ATTRIB_statement> grammar rule, or implicitly using the <fragAttribBinding> grammar rule in an executable instruction.

Each fragment attribute variable is bound to a single item of fragment state according to the <fragAttrBinding> grammar rule. The set of GL state that can be bound to a fragment attribute variable is given in Table X.1. Fragment attribute variables are initialized at each fragment program invocation with the current values of the bound state.

Fragment Attribute Binding	Components	Underlying State
fragment.color	(r,g,b,a)	primary color
fragment.color.primary	(r,g,b,a)	primary color
fragment.color.secondary	(r,g,b,a)	secondary color
fragment.texcoord	(s,t,r,q)	texture coordinate, unit 0
fragment.texcoord[n]	(s,t,r,q)	texture coordinate, unit n
fragment.fogcoord	(f,0,0,1)	fog distance/coordinate
fragment.position	(x,y,z,1/w)	window position

Table X.1: Fragment Attribute Bindings. The "Components" column indicates the mapping of the state in the "Underlying State" column. Bindings containing "[n]" require an integer value of <n> to select an individual item.

If a fragment attribute binding matches "fragment.color" or "fragment.color.primary", the "x", "y", "z", and "w" components of the fragment attribute variable are filled with the "r", "g", "b", and "a" components, respectively, of the fragment color. Each fixed-point color component undergoes an implied conversion to floating point. This conversion must leave the values 0 and 1 invariant.

If a fragment attribute binding matches "fragment.color.secondary", the "x", "y", "z", and "w" components of the fragment attribute variable are filled with the "r", "g", "b", and "a" components, respectively, of the fragment secondary color. Each fixed-point color component undergoes an implied conversion to floating point. This conversion must leave the values 0 and 1 invariant.

If a fragment attribute binding matches "fragment.texcoord" or "fragment.texcoord[n]", the "x", "y", "z", and "w" components of the fragment attribute variable are filled with the "s", "t", "r", and "q" components, respectively, of the fragment texture coordinates for texture unit <n>. If "[n]" is omitted, texture unit zero is used.

If a fragment attribute binding matches "fragment.fogcoord", the "x" component of the fragment attribute variable is filled with either the fragment eye distance or the fog coordinate, depending on whether the fog source is set to FRAGMENT_DEPTH_EXT or FOG_COORDINATE_EXT, respectively. The "y", "z", and "w" coordinates are filled with 0, 0, and 1, respectively.

If a fragment attribute binding matches "fragment.position", the "x" and "y" components of the fragment attribute variable are filled with the (x,y) window coordinates of the fragment center, relative to the lower left corner of the window. The "z" component is filled with the fragment's z window coordinate. This z window coordinate undergoes an implied conversion to floating point. This conversion must leave the values 0 and 1 invariant. The "w" component is filled with the reciprocal of the fragment's clip w coordinate.

On some implementations, the components of fragment.position may be generated by interpolating per-vertex position values. This may produce x and y window coordinates that don't exactly match those of the fragment center and z window coordinates that do not exactly match those generated by fixed-function rasterization. Therefore, there is no guaranteed invariance between the final z window coordinates of fragments processed by fragment programs that write depth values and fragments processed by any other means, even if the fragment programs in question simply copy the z value from the fragment.position binding.

3.11.3.2 Fragment Program Parameters

Fragment program parameter variables are a set of four-component floating-point vectors used as constants during fragment program execution. Fragment program parameters retain their values across fragment program invocations, although their values can change between invocations due to GL state changes.

Single program parameter variables and arrays of program parameter variables can be declared explicitly using the <PARAM_statement> grammar rule. Single program parameter variables can also be declared implicitly using the <paramSingleItemUse> grammar rule in an executable instruction.

Each single program parameter variable is bound to a constant vector or to a GL state vector according to the <paramSingleInit> grammar rule. Individual items of a program parameter array are bound to constant vectors or GL state vectors according to the <programMultipleInit> grammar rule. The set of GL state that can be bound to program parameter variables are given in Tables X.2.1 through X.2.4.

Constant Bindings

A program parameter variable can be bound to a scalar or vector constant using the <paramConstDecl> grammar rule (explicit declarations) or the <paramConstUse> grammar rule (implicit declarations).

If a program parameter binding matches the <paramConstScalarDecl> or <paramConstScalarUse> grammar rules, the corresponding program parameter variable is bound to the vector (X,X,X,X), where X is the value of the specified constant. Note that the <paramConstScalarUse> grammar rule, used only in implicit declarations, allows only non-negative constants. This disambiguates cases like "-2", which could conceivably be taken to mean either the vector "(2,2,2,2)" with all components negated or "(-2,-2,-2,-2)" without negation. Only the former interpretation is allowed by the grammar.

If a program parameter binding matches <paramConstVector>, the corresponding program parameter variable is bound to the vector (X,Y,Z,W), where X, Y, Z, and W are the values corresponding to the first, second, third, and fourth match of <signedFloatConstant>. If fewer than four constants are specified, Y, Z, and W assume the values 0.0, 0.0, and 1.0, if their respective constants are not specified.

Program parameter variables initialized to constant values can never be modified.

Program Environment/Local Parameter Bindings

Binding	Components	Underlying State
program.env[a]	(x,y,z,w)	program environment parameter a
program.local[a]	(x,y,z,w)	program local parameter a
program.env[a..b]	(x,y,z,w)	program environment parameters a through b
program.local[a..b]	(x,y,z,w)	program local parameters a through b

Table X.2.1: Program Environment/Local Parameter Bindings. <a> and <b> indicate parameter numbers, where <a> must be less than or equal to <b>.

If a program parameter binding matches "program.env[a]" or "program.local[a]", the four components of the program parameter variable are filled with the four components of program environment parameter <a> or program local parameter <a>, respectively.

Additionally, for program parameter array bindings, "program.env[a..b]" and "program.local[a..b]" are equivalent to specifying program environment parameters <a> through <b> in order or program local parameters <a> through <b> in order, respectively. In either case, a program will fail to load if <a> is greater than <b>.

Material Property Bindings

Binding	Components	Underlying State
state.material.ambient	(r,g,b,a)	front ambient material color
state.material.diffuse	(r,g,b,a)	front diffuse material color
state.material.specular	(r,g,b,a)	front specular material color
state.material.emission	(r,g,b,a)	front emissive material color
state.material.shininess	(s,0,0,1)	front material shininess
state.material.front.ambient	(r,g,b,a)	front ambient material color
state.material.front.diffuse	(r,g,b,a)	front diffuse material color
state.material.front.specular	(r,g,b,a)	front specular material color
state.material.front.emission	(r,g,b,a)	front emissive material color
state.material.front.shininess	(s,0,0,1)	front material shininess
state.material.back.ambient	(r,g,b,a)	back ambient material color
state.material.back.diffuse	(r,g,b,a)	back diffuse material color
state.material.back.specular	(r,g,b,a)	back specular material color
state.material.back.emission	(r,g,b,a)	back emissive material color
state.material.back.shininess	(s,0,0,1)	back material shininess

Table X.2.2: Material Property Bindings. If a material face is not specified in the binding, the front property is used.

If a program parameter binding matches any of the material properties listed in Table X.2.2, the program parameter variable is filled according to the table. For ambient, diffuse, specular, or emissive colors, the "x", "y", "z", and "w" components are filled with the "r", "g", "b", and "a" components, respectively, of the corresponding material color. For material shininess, the "x" component is filled with the material's specular exponent, and the "y", "z", and "w" components are filled with 0, 0, and 1, respectively. Bindings containing ".back" refer to the back material; all other bindings refer to the front material.

Material properties can be changed inside a Begin/End pair, either directly by calling Material, or indirectly through color material. However, such property changes are not guaranteed to update program parameter bindings until the following End command. Program parameter variables bound to material properties changed inside a Begin/End pair are undefined until the following End command.

Light Property Bindings

Binding	Components	Underlying State
state.light[n].ambient	(r,g,b,a)	light n ambient color
state.light[n].diffuse	(r,g,b,a)	light n diffuse color
state.light[n].specular	(r,g,b,a)	light n specular color
state.light[n].position	(x,y,z,w)	light n position
state.light[n].attenuation	(a,b,c,e)	light n attenuation constants and spot light exponent
state.light[n].spot.direction	(x,y,z,c)	light n spot direction and cutoff angle cosine
state.light[n].half	(x,y,z,1)	light n infinite half-angle
state.lightmodel.ambient	(r,g,b,a)	light model ambient color
state.lightmodel.scenecolor	(r,g,b,a)	light model front scene color
state.lightmodel . front.scenecolor	(r,g,b,a)	light model front scene color
state.lightmodel . back.scenecolor	(r,g,b,a)	light model back scene color
state.lightprod[n].ambient	(r,g,b,a)	light n / front material ambient color product
state.lightprod[n].diffuse	(r,g,b,a)	light n / front material diffuse color product
state.lightprod[n].specular	(r,g,b,a)	light n / front material specular color product
state.lightprod[n]. front.ambient	(r,g,b,a)	light n / front material ambient color product
state.lightprod[n]. front.diffuse	(r,g,b,a)	light n / front material diffuse color product
state.lightprod[n]. front.specular	(r,g,b,a)	light n / front material specular color product
state.lightprod[n]. back.ambient	(r,g,b,a)	light n / back material ambient color product
state.lightprod[n]. back.diffuse	(r,g,b,a)	light n / back material diffuse color product
state.lightprod[n]. back.specular	(r,g,b,a)	light n / back material specular color product

Table X.2.3: Light Property Bindings. <n> indicates a light number.

If a program parameter binding matches "state.light[n].ambient", "state.light[n].diffuse", or "state.light[n].specular", the "x", "y", "z", and "w" components of the program parameter variable are filled with the "r", "g", "b", and "a" components, respectively, of the corresponding light color.

If a program parameter binding matches "state.light[n].position", the "x", "y", "z", and "w" components of the program parameter variable are filled with the "x", "y", "z", and "w" components, respectively, of the light position.

If a program parameter binding matches "state.light[n].attenuation", the "x", "y", and "z" components of the program parameter variable are filled with the constant, linear, and quadratic attenuation parameters of the specified light, respectively (section 2.13.1).

The "w" component of the program parameter variable is filled with the spot light exponent of the specified light.

If a program parameter binding matches "state.light[n].spot.direction", the "x", "y", and "z" components of the program parameter variable are filled with the "x", "y", and "z" components of the spot light direction of the specified light, respectively (section 2.13.1). The "w" component of the program parameter variable is filled with the cosine of the spot light cutoff angle of the specified light.

If a program parameter binding matches "state.light[n].half", the "x", "y", and "z" components of the program parameter variable are filled with the x, y, and z components, respectively, of the normalized infinite half-angle vector

```
  h_inf = || P + (0, 0, 1) ||.
```

The "w" component is filled with 1. In the computation of h_inf, P consists of the x, y, and z coordinates of the normalized vector from the eye position P_e to the eye-space light position P_pli (section 2.13.1). h_inf is defined to correspond to the normalized half-angle vector when using an infinite light (w coordinate of the position is zero) and an infinite viewer (v_bs is FALSE). For local lights or a local viewer, h_inf is well-defined but does not match the normalized half-angle vector, which will vary depending on the vertex position.

If a program parameter binding matches "state.lightmodel.ambient", the "x", "y", "z", and "w" components of the program parameter variable are filled with the "r", "g", "b", and "a" components of the light model ambient color, respectively.

If a program parameter binding matches "state.lightmodel.scenecolor" or "state.lightmodel.front.scenecolor", the "x", "y", and "z" components of the program parameter variable are filled with the "r", "g", and "b" components respectively of the "front scene color"

```
  c_scene = a_cs * a_cm + e_cm,
```

where a_cs is the light model ambient color, a_cm is the front ambient material color, and e_cm is the front emissive material color. The "w" component of the program parameter variable is filled with the alpha component of the front diffuse material color. If a program parameter binding matches "state.lightmodel.back.scenecolor", a similar back scene color, computed using back-facing material properties, is used. The front and back scene colors match the values that would be assigned to vertices using conventional lighting if all lights were disabled.

If a program parameter binding matches anything beginning with "state.lightprod[n]", the "x", "y", and "z" components of the program parameter variable are filled with the "r", "g", and "b" components, respectively, of the corresponding light product. The three light product components are the products of the corresponding color components of the specified material property and the light color of the specified light (see Table X.2.3). The "w" component of the program parameter variable is filled with the alpha component of the specified material property.

Light products depend on material properties, which can be changed inside a Begin/End pair. Such property changes are not guaranteed to take effect until the following End command. Program parameter variables bound to light products whose corresponding material property changes inside a Begin/End pair are undefined until the following End command.

Texture Environment Property Bindings

Binding	Components	Underlying State
state.texenv[n].color	(r,g,b,a)	texture environment n color

Table X.2.4: Texture Environment Property Bindings. "[n]" is optional -- texture unit <n> is used if specified; texture unit 0 is used otherwise.

If a program parameter binding matches "state.texenv[n].color", the "x", "y", "z", and "w" components of the program parameter variable are filled with the "r", "g", "b", and "a" components, respectively, of the corresponding texture environment color. Note that only "legacy" texture units, as queried by MAX_TEXTURE_UNITS, include texture environment state. Texture image units and texture coordinate sets do not have associated texture environment state.

Fog Property Bindings

Binding	Components	Underlying State
state.fog.color	(r,g,b,a)	RGB fog color (section 3.11)
state.fog.params	(d,s,e,r)	fog density, linear start and end, and 1/(end-start) (section 3.11)

Table X.2.5: Fog Property Bindings

If a program parameter binding matches "state.fog.color", the "x", "y", "z", and "w" components of the program parameter variable are filled with the "r", "g", "b", and "a" components, respectively, of the fog color (section 3.11).

If a program parameter binding matches "state.fog.params", the "x", "y", and "z" components of the program parameter variable are filled with the fog density, linear fog start, and linear fog end parameters (section 3.11), respectively. The "w" component is filled with 1/(end-start), where end and start are the linear fog end and start parameters, respectively.

Depth Property Bindings

Binding	Components	Underlying State
state.depth.range	(n,f,d,1)	Depth range near, far, and (far-near) (section 2.10.1)

Table X.2.6: Depth Property Bindings

If a program parameter binding matches "state.depth.range", the "x" and "y" components of the program parameter variable are filled with the mappings of near and far clipping planes to window coordinates, respectively. The "z" component is filled with the difference of the mappings of near and far clipping planes, far minus near. The "w" component is filled with 1.

Matrix Property Bindings

Binding	Underlying State
* state.matrix.modelview[n]	modelview matrix n
state.matrix.projection	projection matrix
state.matrix.mvp	modelview-projection matrix
* state.matrix.texture[n]	texture matrix n
state.matrix.palette[n]	modelview palette matrix n
state.matrix.program[n]	program matrix n

Table X.2.7: Base Matrix Property Bindings. The "[n]" syntax indicates a specific matrix number. For modelview and texture matrices, a matrix number is optional, and matrix zero will be used if the matrix number is omitted. These base bindings may further be modified by a inverse/transpose selector and a row selector.

If the beginning of a program parameter binding matches any of the matrix binding names listed in Table X.2.7, the binding corresponds to a 4x4 matrix. If the parameter binding is followed by ".inverse", ".transpose", or ".invtrans" (<stateMatModifier> grammar rule), the inverse, transpose, or transpose of the inverse, respectively, of the matrix specified in Table X.2.7 is selected. Otherwise, the matrix specified in Table X.2.7 is selected. If the specified matrix is poorly-conditioned (singular or nearly so), its inverse matrix is undefined. The binding name "state.matrix.mvp" refers to the product of modelview matrix zero and the projection matrix, defined as

```
 MVP = P * M0,
```

where P is the projection matrix and M0 is modelview matrix zero.

If the selected matrix is followed by ".row[<a>]" (matching the <stateMatrixRow> grammar rule), the "x", "y", "z", and "w" components of the program parameter variable are filled with the four entries of row <a> of the selected matrix. In the example,

```
PARAM m0 = state.matrix.modelview[1].row[0];
PARAM m1 = state.matrix.projection.transpose.row[3];
```

the variable "m0" is set to the first row (row 0) of modelview matrix 1 and "m1" is set to the last row (row 3) of the transpose of the projection matrix.

For program parameter array bindings, multiple rows of the selected matrix can be bound via the <stateMatrixRows> grammar rule. If the selected matrix binding is followed by ".row[<a>..<b>]", the result is equivalent to specifying matrix rows <a> through <b>, in order. A program will fail to load if <a> is greater than <b>. If no row selection is specified (<optMatrixRows> matches ""), matrix rows 0 through 3 are bound in order. In the example,

```
  PARAM m2[] = { state.matrix.program[0].row[1..2] };
  PARAM m3[] = { state.matrix.program[0].transpose };
```

the array "m2" has two entries, containing rows 1 and 2 of program matrix zero, and "m3" has four entries, containing all four rows of the transpose of program matrix zero.

Program Parameter Arrays

A program parameter array variable can be declared explicitly by matching the <PARAM_multipleStmt> grammar rule. Programs can optionally specify the number of individual program parameters in the array, using the <optArraySize> grammar rule. Program parameter arrays may not be declared implicity.

Individual parameter variables in a program parameter array are bound to GL state vectors or constant vectors as specified by the grammar rule <paramMultInitList>. Each individual parameter in the array is bound in turn as described above.

The total number of entries in the array is equal to the number of parameters bound in the initializer list. A fragment program that specifies an array size (<optArraySize> matches <integer>) that does not match the number of parameter bindings in the initialization list will fail to load.

Program parameter array variables may only be accessed using absolute addressing by matching the <progParamArrayAbs> grammar rule. Array accesses are checked against the limits of the array. If any fragment program instruction accesses a program parameter array with an out-of-range index (greater than or equal to the size of the array), the fragment program will fail to load.

Individual state vectors can have no more than one unique binding in any given program. The GL will automatically combine multiple bindings of the same state vector into a single unique binding.

3.11.3.3 Fragment Program Temporaries

Fragment program temporary variables are a set of four-component floating-point vectors used to hold temporary results during fragment program execution. Temporaries do not persist between program invocations, and are undefined at the beginning of each fragment program invocation.

Fragment program temporary variables can be declared explicitly using the <TEMP_statement> grammar rule. Each such statement can declare one or more temporaries. Fragment program temporary variables can not be declared implicitly.

3.11.3.4 Fragment Program Results

Fragment program result variables are a set of four component floating-point vectors used to hold the final results of a fragment program. Fragment program result variables are write-only during fragment program execution.

Fragment program result variables can be declared explicitly using the <OUTPUT_statement> grammar rule, or implicitly using the <resultBinding> grammar rule in an executable instruction. Each fragment program result variable is bound to a fragment attribute used in subsequent back-end processing. The set of fragment program result variable bindings is given in Table X.3.

Binding	Components	Description
result.color	(r,g,b,a)	color
result.depth	(*,*,d,*)	depth coordinate

Table X.3: Fragment Result Variable Bindings. Components labeled "*" are unused.

If a result variable binding matches "result.color", updates to the "x", "y", "z", and "w" components of the result variable modify the "r", "g", "b", and "a" components, respectively, of the fragment's output color. If "result.color" is not both bound by the fragment program and written by some instruction of the program, the output color of the fragment program is undefined.

If a result variable binding matches "result.depth", updates to the "z" component of the result variable modify the fragment's output depth value. If "result.depth" is not both bound by the fragment program and written by some instruction of the program, the interpolated depth value produced by rasterization is used as if fragment program mode is not enabled. Writes to any component of depth other than the "z" component have no effect.

3.11.3.5 Fragment Program Aliases

Fragment programs can create aliases by matching the <ALIAS_statement> grammar rule. Aliases allow programs to use multiple variable names to refer to a single underlying variable. For example, the statement

```
ALIAS var1 = var0
```

establishes a variable name named "var1". Subsequent references to "var1" in the program text are treated as references to "var0". The left hand side of an ALIAS statement must be a new variable name, and the right hand side must be an established variable name.

Aliases are not considered variable declarations, so do not count against the limits on the number of variable declarations allowed in the program text.

3.11.3.6 Fragment Program Resource Limits

The fragment program execution environment provides implementation-dependent resource limits on the number of ALU instructions, texture instructions, total instructions (ALU or texture), temporary variable declarations, program parameter bindings, or texture indirections. A program that exceeds any of these resource limits will fail to load. The resource limits for fragment programs can be queried by calling GetProgramiv (section 6.1.12) with a target of FRAGMENT_PROGRAM_ARB.

The limit on fragment program ALU instructions can be queried with a <pname> of MAX_PROGRAM_ALU_INSTRUCTIONS_ARB, and must be at least 48. Each ALU instruction in the program (matches of the <ALUInstruction> grammar rule) counts against this limit.

The limit on fragment program texture instructions can be queried with a <pname> of MAX_PROGRAM_TEX_INSTRUCTIONS_ARB, and must be at least 24. Each texture instruction in the program (matches of the <TexInstruction> grammar rule) counts against this limit.

The limit on fragment program total instructions can be queried with a <pname> of MAX_PROGRAM_INSTRUCTIONS_ARB, and must be at least 72. Each instruction in the program (matching the <instruction> grammar rule) counts against this limit. Note that the limit on total instructions is not necessarily equal to the sum of the limits on ALU instructions and texture instructions.

The limit on fragment program texture indirections can be queried with a <pname> of MAX_PROGRAM_TEX_INDIRECTIONS_ARB, and must be at least 4. Texture indirections are described in 3.11.6. If an implementation has no limit on texture indirections, the limit will be equal to the limit on texture instructions.

The limit on fragment program temporary variable declarations can be queried with a <pname> of MAX_PROGRAM_TEMPORARIES_ARB, and must be at least 16. Each temporary declared in the program, using the <TEMP_statement> grammar rule, counts against this limit. Aliases of declared temporaries do not.

The limit on fragment program attribute bindings can be queried with a <pname> of MAX_PROGRAM_ATTRIBS_ARB and must be at least 10. Each distinct vertex attribute bound explicitly or implicitly in the program counts against this limit; vertex attributes bound multiple times count only once.

The limit on fragment program parameter bindings can be queried with a <pname> of MAX_PROGRAM_PARAMETERS_ARB, and must be at least 24. Each distinct GL state vector bound explicitly or implicitly in the program counts against this limit; GL state vectors bound multiple times count only once. Every other constant vector bound in the program is counted if and only if an identical constant vector has not already been counted. Two constant vectors are considered identical if the four component values are numerically equivalent. Recall that scalar constants bound in a program are treated as vector constants with the scalar value replicated.

In addition to the limits described above, the GL provides a similar set of implementation-dependent native resource limits. These limits, specified in Section 6.1.12, provide guidance as to whether the program is small enough to use a "native" mode where fragment programs may be executed with higher performance. The native resource limits and usage counts are implementation-dependent and may not exactly correspond to limits and counts described above. A program's native resource consumption may be reduced by program optimizations performed by the GL. Native resource consumption may be increased due to emulation of instructions or any other program features not natively supported by an implementation. Notably, an additional texture indirection may be consumed due to an implementation's lack of native support for texture instructions with source coordinate swizzles or parameter source coordinates, which may require emulation by prepending ALU instructions. An implementation may also fail to natively support all combinations of attributes described in Table X.1, even if the total number of bound attributes is fewer than the native attribute limit. In this case the program is still considered to exceed the native resource limits, as queried by PROGRAM_UNDER_NATIVE_LIMITS_ARB (section 6.1.12).

To assist in resource counting, the GL additionally provides GetProgram queries to determine the resource usage and native resource usage of the currently bound program, and to determine whether the bound program exceeds any native resource limit.

Programs that exceed any native resource limit may or may not load depending on the implementation.

3.11.4 Fragment Program Execution Environment

If fragment program mode is enabled, the currently bound fragment program is executed when any fragment is produced by rasterization.

If fragment program mode is enabled and the currently bound program object does not contain a valid fragment program, the error INVALID_OPERATION will be generated by Begin, RasterPos, and any command that implicitly calls Begin (e.g., DrawArrays).

Fragment programs execute a sequence of instructions without branching. Fragment programs begin by executing the first instruction in the program, and execute instructions in the order specified in the program until the last instruction is completed.

There are 33 fragment program instructions. The instructions and their respective input and output parameters are summarized in Table X.5.

Instruction	Inputs	Output	Description
ABS	v	v	absolute value
ADD	v,v	v	add
CMP	v,v,v	v	compare
COS	s	ssss	cosine with reduction to [-PI,PI]
DP3	v,v	ssss	3-component dot product
DP4	v,v	ssss	4-component dot product
DPH	v,v	ssss	homogeneous dot product
DST	v,v	v	distance vector
EX2	s	ssss	exponential base 2
FLR	v	v	floor
FRC	v	v	fraction
KIL	v	v	kill fragment
LG2	s	ssss	logarithm base 2
LIT	v	v	compute light coefficients
LRP	v,v,v	v	linear interpolation
MAD	v,v,v	v	multiply and add
MAX	v,v	v	maximum
MIN	v,v	v	minimum
MOV	v	v	move
MUL	v,v	v	multiply
POW	s,s	ssss	exponentiate
RCP	s	ssss	reciprocal
RSQ	s	ssss	reciprocal square root
SCS	s	ss--	sine/cosine without reduction
SGE	v,v	v	set on greater than or equal
SIN	s	ssss	sine with reduction to [-PI,PI]
SLT	v,v	v	set on less than
SUB	v,v	v	subtract
SWZ	v	v	extended swizzle
TEX	v,u,t	v	texture sample
TXB	v,u,t	v	texture sample with bias
TXP	v,u,t	v	texture sample with projection
XPD	v,v	v	cross product

Table X.5: Summary of fragment program instructions. "v" indicates a floating-point vector input or output, "s" indicates a floating-point scalar input, "ssss" indicates a scalar output replicated across a 4-component result vector, "ss--" indicates two scalar outputs in the first two components, "u" indicates a texture image unit identifier, and "t" indicates a texture target.

3.11.4.1 Fragment Program Operands

Most fragment program instructions operate on floating-point vectors or scalars, as indicated by the grammar rules <vectorSrcReg> and <scalarSrcReg>, respectively.

Vector and scalar operands can be obtained from fragment attribute, program parameter, or temporary registers, as indicated by the <srcReg> rule. For scalar operands, a single vector component is selected by the <scalarSuffix> rule, where the characters "x", "y", "z", and "w", or "r", "g", "b", and "a" select the first, second, third, and fourth components, respectively, of the vector.

Vector operands can be swizzled according to the <optionalSuffix> rule. In its most general form, the <optionalSuffix> rule matches the pattern ".????" where each question mark is replaced with one of "x", "y", "z", "w", "r", "g", "b", or "a". For such patterns, the first, second, third, and fourth components of the operand are taken from the vector components named by the first, second, third, and fourth character of the pattern, respectively. For example, if the swizzle suffix is ".yzzx" or ".gbbr" and the specified source contains {2,8,9,0}, the swizzled operand used by the instruction is {8,9,9,2}.

If the <optionalSuffix> rule matches "", it is treated as though it were ".xyzw". If the <optionalSuffix> rule matches (ignoring whitespace) ".x", ".y", ".z", or ".w", these are treated the same as ".xxxx", ".yyyy", ".zzzz", and ".wwww" respectively. Likewise, if the <optionalSuffix> rule matches ".r", ".g", ".b", or ".a", these are treated the same as ".rrrr", ".gggg", ".bbbb", and ".aaaa" respectively.

Floating-point scalar or vector operands can optionally be negated according to the <optionalSign> rule in <scalarSrcReg> and <vectorSrcReg>. If the <optionalSign> matches "-", each operand or operand component is negated.

The following pseudo-code spells out the operand generation process. In the example, "float" is a floating-point scalar type, while "floatVec" is a four-component vector. "source" refers to the register used for the operand, matching the <srcReg> rule. "negate" is TRUE if the <optionalSign> rule in <scalarSrcReg> or <vectorSrcReg> matches "-" and FALSE otherwise. The ".c***", ".*c**", ".**c*", ".***c" modifiers refer to the x, y, z, and w components obtained by the swizzle operation; the ".c" modifier refers to the single component selected for a scalar load.

```
floatVec VectorLoad(floatVec source)
{
    floatVec operand;

    operand.x = source.c***;
    operand.y = source.*c**;
    operand.z = source.**c*;
    operand.w = source.***c;
    if (negate) {
       operand.x = -operand.x;
       operand.y = -operand.y;
       operand.z = -operand.z;
       operand.w = -operand.w;
    }

    return operand;
}
```

```
  float ScalarLoad(floatVec source)
  {
      float operand;

      operand = source.c;
      if (negate) {
        operand = -operand;
      }

      return operand;
  }
```

3.11.4.2 Fragment Program Parameter Arrays

A fragment program can load a single element of a program parameter array using only absolute addressing. Program parameter arrays are accessed when the <progParamArrayAbs> rule is matched. The offset of the selected entry in the array is given by the number matching <progParamRegNum>. If the offset exceeds the size of the array, the results of the access are undefined, but may not lead to program or GL termination.

3.11.4.3 Fragment Program Destination Register Update

Fragment program instructions write a 4-component result vector to a single temporary or fragment result register. Writes to individual components of the destination register are controlled by individual component write masks specified as part of the instruction. Optional clamping of each component of the destination register to the range [0,1] is controlled by an opcode modifier.

The component write mask is specified by the <optionalMask> rule found in the <maskedDstReg> rule. If the optional mask is "", all components are enabled. Otherwise, the optional mask names the individual components to enable. The characters "x", "y", "z", and "w", or "r", "g", "b", and "a" match the first, second, third, and fourth components, respectively. For example, an optional mask of ".xzw" indicates that the x, z, and w components should be enabled for writing but the y component should not. The grammar requires that the destination register mask components must be listed in "xyzw", or "rgba" order. Component names from one set (xyzw or rgba) cannot be mixed with component names from another set. For example, ".rgw" is not a valid writemask.

Each component of the destination register is updated with the result of the fragment program instruction if and only if the component is enabled for writes by the component write mask. Otherwise, the component of the destination register remains unchanged.

If the instruction opcode has the "_SAT" suffix, requesting saturated result vectors, each component of the result vector enabled in the writemask is clamped to the range [0,1] before being updated in the destination register.

The following pseudocode illustrates the process of writing a result vector to the destination register. In the pseudocode, "instrmask"

refers to the component write mask given by the <optionalMask> rule.
"clamp" is TRUE if the instruction specifies that the result should
be clamped. "result" and "destination" refer to the result vector
and the register selected by <dstReg>, respectively.

```
  void UpdateDestination(floatVec destination, floatVec result)
  {
      floatVec merged;

      // Clamp the result vector components to [0,1], if requested.
      if (instrClamp) {
          if (result.x < 0)      result.x = 0;
          else if (result.x > 1) result.x = 1;
          if (result.y < 0)      result.y = 0;
          else if (result.y > 1) result.y = 1;
          if (result.z < 0)      result.z = 0;
          else if (result.z > 1) result.z = 1;
          if (result.w < 0)      result.w = 0;
          else if (result.w > 1) result.w = 1;
      }

      // Merge the converted result into the destination register,
      // under control of the compile-time write mask.
      merged = destination;
      if (instrMask.x) {
          merged.x = result.x;
      }
      if (instrMask.y) {
          merged.y = result.y;
      }
      if (instrMask.z) {
          merged.z = result.z;
      }
      if (instrMask.w) {
          merged.w = result.w;
      }

      // Write out the new destination register.
      destination = merged;
  }
```

3.11.4.4 Fragment Program Result Processing

As a fragment program executes, it will write to either one or two
result registers that are mapped to the fragment's color and depth.

The fragment's color components are first clamped to the range [0,1]
then converted to fixed point as in section 2.13.9. If the fragment
program does not write result.color, the color will be undefined in
subsequent stages.

If the fragment program contains an instruction to write to
result.depth, the fragment's depth is replaced by the value of the
"z" component of result.depth. This z value is first clamped to the
range [0,1] then converted to fixed-point as if it were a window z
value (section 2.10.1). If the fragment program does not write
result.depth, the fragment's original depth is unmodified.

3.11.4.5 Fragment Program Options

The <optionSequence> grammar rule provides a mechanism for programs to indicate that one or more extended language features are used by the program. All program options used by the program must be declared at the beginning of the program string. Each program option specified in a program string will modify the syntactic or semantic rules used to interpet the program and the execution environment used to execute the program. Program options not present in the program string are ignored, even if they are supported by the GL.

The <identifier> token in the <option> rule must match the name of a program option supported by the implementation. To avoid option name conflicts, option identifiers are required to begin with a vendor prefix. A program will fail to load if it specifies a program option not supported by the GL.

Fragment program options should confine their semantic changes to the domain of fragment programs. Support for a fragment program option should not change the specification and behavior of fragment programs not requesting use of that option.

3.11.4.5.1 Fog Application Fragment Program Options

If a fragment program specifies one of the options "ARB_fog_exp", "ARB_fog_exp2", or "ARB_fog_linear", the program will apply fog to the program's final clamped color using a fog mode of EXP, EXP2, or LINEAR, respectively, as described in section 3.10.

When a fog option is specified in a fragment program, semantic restrictions are added to indicate that a fragment program will fail to load if the number of temporaries it contains exceeds the implementation-dependent limit minus 1, if the number of attributes it contains exceeds the implementation-dependent limit minus 1, or if the number of parameters it contains exceeds the implementation-dependent limit minus 2.

Additionally, when the ARB_fog_exp option is specified in a fragment program, a semantic restriction is added to indicate that a fragment program will fail to load if the number of instructions or ALU instructions it contains exceeds the implementation-dependent limit minus 3. When the ARB_fog_exp2 option is specified in a fragment program, a semantic restriction is added to indicate that a fragment program will fail to load if the number of instructions or ALU instructions it contains exceeds the implementation-dependent limit minus 4. When the ARB_fog_linear option is specified in a fragment program, a semantic restriction is added to indicate that a fragment program will fail to load if the number of instructions or ALU instructions it contains exceeds the implementation-dependent limit minus 2.

Only one fog application option may be specified by any given fragment program. A fragment program that specifies more than one of the program options "ARB_fog_exp", "ARB_fog_exp2", and "ARB_fog_linear", will fail to load.

3.11.4.5.2 Precision Hint Options

Fragment program computations are carried out at an implementation-dependent precision. However, some implementations may be able to perform fragment program computations at more than one precision, and may be able to trade off computation precision for performance.

If a fragment program specifies the "ARB_precision_hint_fastest" program option, implementations should select precision to minimize program execution time, with possibly reduced precision. If a fragment program specifies the "ARB_precision_hint_nicest" program option, implementations should maximize the precision, with possibly increased execution time.

Only one precision control option may be specified by any given fragment program. A fragment program that specifies both the "ARB_precision_hint_fastest" and "ARB_precision_hint_nicest" program options will fail to load.

3.11.5 Fragment Program ALU Instruction Set

The following sections describe the set of supported fragment program instructions. Each section contains pseudocode describing the instruction. Instructions will have up to three operands, referred to as "op0", "op1", and "op2". The operands are loaded using the mechanisms specified in section 3.11.4.1. The variables "tmp", "tmp0", "tmp1", and "tmp2" describe scalars or vectors used to hold intermediate results in the instruction. Instructions will generate a result vector called "result". The result vector is then written to the destination register specified in the instruction as described in section 3.11.4.3.

3.11.5.1 ABS: Absolute Value

The ABS instruction performs a component-wise absolute value operation on the single operand to yield a result vector.

```
  tmp = VectorLoad(op0);
  result.x = fabs(tmp.x);
  result.y = fabs(tmp.y);
  result.z = fabs(tmp.z);
  result.w = fabs(tmp.w);
```

3.11.5.2 ADD: Add

The ADD instruction performs a component-wise add of the two operands to yield a result vector.

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1);
result.x = tmp0.x + tmp1.x;
result.y = tmp0.y + tmp1.y;
result.z = tmp0.z + tmp1.z;
result.w = tmp0.w + tmp1.w;
```

The following rules apply to addition:

```
1. <x> + <y> == <y> + <x>, for all <x> and <y>.
2. <x> + 0.0 == <x>, for all <x>.
```

3.11.5.3 CMP: Compare

The CMP instructions performs a component-wise comparison of the first operand against zero, and copies the values of the second or third operands based on the results of the compare.

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1);
tmp2 = VectorLoad(op2);
result.x = (tmp0.x < 0.0) ? tmp1.x : tmp2.x;
result.y = (tmp0.y < 0.0) ? tmp1.y : tmp2.y;
result.z = (tmp0.z < 0.0) ? tmp1.z : tmp2.z;
result.w = (tmp0.w < 0.0) ? tmp1.w : tmp2.w;
```

3.11.5.4 COS: Cosine

The COS instruction approximates the trigonometric cosine of the angle specified by the scalar operand and replicates it to all four components of the result vector. The angle is specified in radians and does not have to be in the range [-PI,PI].

```
tmp = ScalarLoad(op0);
result.x = ApproxCosine(tmp);
result.y = ApproxCosine(tmp);
result.z = ApproxCosine(tmp);
result.w = ApproxCosine(tmp);
```

3.11.5.5 DP3: Three-Component Dot Product

The DP3 instruction computes a three-component dot product of the two operands (using the first three components) and replicates the dot product to all four components of the result vector.

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1);
dot = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) + (tmp0.z * tmp1.z);
result.x = dot;
result.y = dot;
result.z = dot;
result.w = dot;
```

3.11.5.6 DP4: Four-Component Dot Product

The DP4 instruction computes a four-component dot product of the two operands and replicates the dot product to all four components of the result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1):
  dot = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
        (tmp0.z * tmp1.z) + (tmp0.w * tmp1.w);
  result.x = dot;
  result.y = dot;
  result.z = dot;
  result.w = dot;
```

3.11.5.7 DPH: Homogeneous Dot Product

The DPH instruction computes a three-component dot product of the two operands (using the x, y, and z components), adds the w component of the second operand, and replicates the sum to all four components of the result vector. This is equivalent to a four-component dot product where the w component of the first operand is forced to 1.0.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1):
  dot = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
        (tmp0.z * tmp1.z) + tmp1.w;
  result.x = dot;
  result.y = dot;
  result.z = dot;
  result.w = dot;
```

3.11.5.8 DST: Distance Vector

The DST instruction computes a distance vector from two specially-formatted operands. The first operand should be of the form [NA, d^2, d^2, NA] and the second operand should be of the form [NA, 1/d, NA, 1/d], where NA values are not relevant to the calculation and d is a vector length. If both vectors satisfy these conditions, the result vector will be of the form [1.0, d, d^2, 1/d].

The exact behavior is specified in the following pseudo-code:

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = 1.0;
  result.y = tmp0.y * tmp1.y;
  result.z = tmp0.z;
  result.w = tmp1.w;
```

Given an arbitrary vector, d^2 can be obtained using the DP3 instruction (using the same vector for both operands) and 1/d can be obtained from d^2 using the RSQ instruction.

This distance vector is useful for per-fragment light attenuation calculations: a DP3 operation using the distance vector and an attenuation constants vector as operands will yield the attenuation factor.

3.11.5.9 EX2: Exponential Base 2

The EX2 instruction approximates 2 raised to the power of the scalar operand and replicates the approximation to all four components of the result vector.

```
  tmp = ScalarLoad(op0);
  result.x = Approx2ToX(tmp);
  result.y = Approx2ToX(tmp);
  result.z = Approx2ToX(tmp);
  result.w = Approx2ToX(tmp);
```

3.11.5.10 FLR: Floor

The FLR instruction performs a component-wise floor operation on the operand to generate a result vector. The floor of a value is defined as the largest integer less than or equal to the value. The floor of 2.3 is 2.0; the floor of -3.6 is -4.0.

```
  tmp = VectorLoad(op0);
  result.x = floor(tmp.x);
  result.y = floor(tmp.y);
  result.z = floor(tmp.z);
  result.w = floor(tmp.w);
```

3.11.5.11 FRC: Fraction

The FRC instruction extracts the fractional portion of each component of the operand to generate a result vector. The fractional portion of a component is defined as the result after subtracting off the floor of the component (see FLR), and is always in the range [0.0, 1.0).

For negative values, the fractional portion is NOT the number written to the right of the decimal point -- the fractional portion of -1.7 is not 0.7 -- it is 0.3. 0.3 is produced by subtracting the floor of -1.7 (-2.0) from -1.7.

```
  tmp = VectorLoad(op0);
  result.x = fraction(tmp.x);
  result.y = fraction(tmp.y);
  result.z = fraction(tmp.z);
  result.w = fraction(tmp.w);
```

3.11.5.12 LG2: Logarithm Base 2

The LG2 instruction approximates the base 2 logarithm of the scalar operand and replicates it to all four components of the result vector.

```
  tmp = ScalarLoad(op0);
  result.x = ApproxLog2(tmp);
  result.y = ApproxLog2(tmp);
  result.z = ApproxLog2(tmp);
  result.w = ApproxLog2(tmp);
```

If the scalar operand is zero or negative, the result is undefined.

3.11.5.13 LIT: Light Coefficients

The LIT instruction accelerates per-fragment lighting by computing lighting coefficients for ambient, diffuse, and specular light contributions. The "x" component of the single operand is assumed to hold a diffuse dot product (n dot VP_pli, as in the vertex lighting equations in Section 2.13.1). The "y" component of the operand is assumed to hold a specular dot product (n dot h_i). The "w" component of the operand is assumed to hold the specular exponent of the material (s_rm), and is clamped to the range (-128, +128) exclusive.

The "x" component of the result vector receives the value that should be multiplied by the ambient light/material product (always 1.0). The "y" component of the result vector receives the value that should be multiplied by the diffuse light/material product (n dot VP_pli). The "z" component of the result vector receives the value that should be multiplied by the specular light/material product (f_i * (n dot h_i) ^ s_rm). The "w" component of the result is the constant 1.0.

Negative diffuse and specular dot products are clamped to 0.0, as is done in the standard per-vertex lighting operations. In addition, if the diffuse dot product is zero or negative, the specular coefficient is forced to zero.

```
  tmp = VectorLoad(op0);
  if (tmp.x < 0) tmp.x = 0;
  if (tmp.y < 0) tmp.y = 0;
  if (tmp.w < -(128.0-epsilon)) tmp.w = -(128.0-epsilon);
  else if (tmp.w > 128-epsilon) tmp.w = 128-epsilon;
  result.x = 1.0;
  result.y = tmp.x;
  result.z = (tmp.x > 0) ? RoughApproxPower(tmp.y, tmp.w) : 0.0;
  result.w = 1.0;
```

The exponentiation approximation function may be defined in terms of the base 2 exponentiation and logarithm approximation operations in the EX2 and LG2 instructions, where

```
  ApproxPower(a,b) = ApproxExp2(b * ApproxLog2(a)).
```

In particular, the approximation may not be any more accurate than the underlying EX2 and LG2 operations.

Also, since 0^0 is defined to be 1, RoughApproxPower(0.0, 0.0) will produce 1.0.

3.11.5.14 LRP: Linear Interpolation

The LRP instruction performs a component-wise linear interpolation between the second and third operands using the first operand as the blend factor.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  tmp2 = VectorLoad(op2);
  result.x = tmp0.x * tmp1.x + (1 - tmp0.x) * tmp2.x;
  result.y = tmp0.y * tmp1.y + (1 - tmp0.y) * tmp2.y;
  result.z = tmp0.z * tmp1.z + (1 - tmp0.z) * tmp2.z;
  result.w = tmp0.w * tmp1.w + (1 - tmp0.w) * tmp2.w;
```

3.11.5.15 MAD: Multiply and Add

The MAD instruction performs a component-wise multiply of the first two operands, and then does a component-wise add of the product to the third operand to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  tmp2 = VectorLoad(op2);
  result.x = tmp0.x * tmp1.x + tmp2.x;
  result.y = tmp0.y * tmp1.y + tmp2.y;
  result.z = tmp0.z * tmp1.z + tmp2.z;
  result.w = tmp0.w * tmp1.w + tmp2.w;
```

The multiplication and addition operations in this instruction are subject to the same rules as described for the MUL and ADD instructions.

3.11.5.16 MAX: Maximum

The MAX instruction computes component-wise maximums of the values in the two operands to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x > tmp1.x) ? tmp0.x : tmp1.x;
  result.y = (tmp0.y > tmp1.y) ? tmp0.y : tmp1.y;
  result.z = (tmp0.z > tmp1.z) ? tmp0.z : tmp1.z;
  result.w = (tmp0.w > tmp1.w) ? tmp0.w : tmp1.w;
```

3.11.5.17 MIN: Minimum

The MIN instruction computes component-wise minimums of the values in the two operands to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x > tmp1.x) ? tmp1.x : tmp0.x;
  result.y = (tmp0.y > tmp1.y) ? tmp1.y : tmp0.y;
  result.z = (tmp0.z > tmp1.z) ? tmp1.z : tmp0.z;
  result.w = (tmp0.w > tmp1.w) ? tmp1.w : tmp0.w;
```

3.11.5.18 MOV: Move

The MOV instruction copies the value of the operand to yield a result vector.

```
  result = VectorLoad(op0);
```

3.11.5.19 MUL: Multiply

The MUL instruction performs a component-wise multiply of the two operands to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = tmp0.x * tmp1.x;
  result.y = tmp0.y * tmp1.y;
  result.z = tmp0.z * tmp1.z;
  result.w = tmp0.w * tmp1.w;
```

The following rules apply to multiplication:

1. <x> * <y> == <y> * <x>, for all <x> and <y>.
2. +/-0.0 * <x> = +/-0.0, at least for all <x> that correspond to representable numbers (IEEE "not a number" and "infinity" encodings may be exceptions).
3. +1.0 * <x> = <x>, for all <x>.

Multiplication by zero and one should be invariant, as it may be used to evaluate conditional expressions without branching.

3.11.5.20 POW: Exponentiate

The POW instruction approximates the value of the first scalar operand raised to the power of the second scalar operand and replicates it to all four components of the result vector.

```
  tmp0 = ScalarLoad(op0);
  tmp1 = ScalarLoad(op1);
  result.x = ApproxPower(tmp0, tmp1);
  result.y = ApproxPower(tmp0, tmp1);
  result.z = ApproxPower(tmp0, tmp1);
  result.w = ApproxPower(tmp0, tmp1);
```

The exponentiation approximation function may be implemented using the base 2 exponentiation and logarithm approximation operations in the EX2 and LG2 instructions. In particular,

```
  ApproxPower(a,b) = ApproxExp2(b * ApproxLog2(a)).
```

Note that a logarithm may be involved even for cases where the exponent is an integer. This means that it may not be possible to exponentiate correctly with a negative base. In constrast, it is possible in a "normal" mathematical formulation to raise negative numbers to integral powers (e.g., (-3)^2== 9, and (-0.5)^-2==4).

3.11.5.21 RCP: Reciprocal

The RCP instruction approximates the reciprocal of the scalar operand and replicates it to all four components of the result vector.

```
  tmp = ScalarLoad(op0);
  result.x = ApproxReciprocal(tmp);
  result.y = ApproxReciprocal(tmp);
  result.z = ApproxReciprocal(tmp);
  result.w = ApproxReciprocal(tmp);
```

The following rule applies to reciprocation:

```
  1. ApproxReciprocal(+1.0) = +1.0.
```

3.11.5.22 RSQ: Reciprocal Square Root

The RSQ instruction approximates the reciprocal of the square root of the absolute value of the scalar operand and replicates it to all four components of the result vector.

```
  tmp = fabs(ScalarLoad(op0));
  result.x = ApproxRSQRT(tmp);
  result.y = ApproxRSQRT(tmp);
  result.z = ApproxRSQRT(tmp);
  result.w = ApproxRSQRT(tmp);
```

3.11.5.23 SCS: Sine/Cosine

The SCS instruction approximates the trigonometric sine and cosine of the angle specified by the scalar operand and places the cosine in the x component and the sine in the y component of the result vector. The z and w components of the result vector are undefined. The angle is specified in radians and must be in the range [-PI,PI].

```
  tmp = ScalarLoad(op0);
  result.x = ApproxCosine(tmp);
  result.y = ApproxSine(tmp);
```

If the scalar operand is not in the range [-PI,PI], the result vector is undefined.

3.11.5.24 SGE: Set On Greater or Equal Than

The SGE instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operands is greater than or equal that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x >= tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y >= tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z >= tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w >= tmp1.w) ? 1.0 : 0.0;
```

3.11.5.25 SIN: Sine

The SIN instruction approximates the trigonometric sine of the angle specified by the scalar operand and replicates it to all four components of the result vector. The angle is specified in radians and does not have to be in the range [-PI,PI].

```
  tmp = ScalarLoad(op0);
  result.x = ApproxSine(tmp);
  result.y = ApproxSine(tmp);
  result.z = ApproxSine(tmp);
  result.w = ApproxSine(tmp);
```

3.11.5.26 SLT: Set On Less Than

The SLT instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operand is less than that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x < tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y < tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z < tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w < tmp1.w) ? 1.0 : 0.0;
```

3.11.5.27 SUB: Subtract

The SUB instruction performs a component-wise subtraction of the second operand from the first to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = tmp0.x - tmp1.x;
  result.y = tmp0.y - tmp1.y;
  result.z = tmp0.z - tmp1.z;
  result.w = tmp0.w - tmp1.w;
```

3.11.5.28 SWZ: Extended Swizzle

The SWZ instruction loads the single vector operand, and performs a swizzle operation more powerful than that provided for loading normal vector operands to yield an instruction vector.

After the operand is loaded, the "x", "y", "z", and "w" components of the result vector are selected by the first, second, third, and fourth matches of the <xyzwExtSwizComp> or <rgbaExtSwizComp> pattern in the <extendedSwizzle> rule.

A result component can be selected from any of the four components of the operand or the constants 0.0 and 1.0. The result component can also be optionally negated. The following pseudocode describes the component selection method. "operand" refers to the vector operand. "select" is an enumerant where the values ZERO, ONE, X, Y, Z, and W correspond to the <xyzwExtSwizSel> rule matching "0", "1", "x", "y", "z", and "w", respectively, or the <rgbaExtSwizSel> rule matching "0", 1", "r", "g", "b", and "a", respectively. "negate" is TRUE if and only if the <optionalSign> rule in <xyzwExtSwizComp> or <rgbaExtSwizComp> matches "-".

```
float ExtSwizComponent(floatVec operand, enum select, boolean negate)
{
    float result;
    switch (select) {
      case ZERO:  result = 0.0; break;
      case ONE:   result = 1.0; break;
      case X:     result = operand.x; break;
      case Y:     result = operand.y; break;
      case Z:     result = operand.z; break;
      case W:     result = operand.w; break;
    }
    if (negate) {
      result = -result;
    }
    return result;
}
```

The entire extended swizzle operation is then defined using the following pseudocode:

```
tmp = VectorLoad(op0);
result.x = ExtSwizComponent(tmp, xSelect, xNegate);
result.y = ExtSwizComponent(tmp, ySelect, yNegate);
result.z = ExtSwizComponent(tmp, zSelect, zNegate);
result.w = ExtSwizComponent(tmp, wSelect, wNegate);
```

"xSelect", "xNegate", "ySelect", "yNegate", "zSelect", "zNegate", "wSelect", and "wNegate" correspond to the "select" and "negate" values above for the four <xyzwExtSwizComp> or <rgbaExtSwizComp> matches.

Since this instruction allows for component selection and negation for each individual component, the grammar does not allow the use of the normal swizzle and negation operations allowed for vector operands in other instructions.

3.11.5.29 XPD: Cross Product

The XPD instruction computes the cross product using the first three components of its two vector operands to generate the x, y, and z components of the result vector. The w component of the result vector is undefined.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = tmp0.y * tmp1.z - tmp0.z * tmp1.y;
  result.y = tmp0.z * tmp1.x - tmp0.x * tmp1.z;
  result.z = tmp0.x * tmp1.y - tmp0.y * tmp1.x;
```

3.11.6 Fragment Program Texture Instruction Set

The first three texture instructions described below specify the mapping of 4-tuple vectors to colors of an image. The sampling of the texture works as described in section 3.8, except that texture environments and texture functions are not applicable, and the texture enables hierarchy is replaced by explicit references to the desired texture target (i.e., 1D, 2D, 3D, cube map, rectangle). These texture instructions specify how the 4-tuple is mapped into the coordinates used for sampling. The following function is used to describe the texture sampling in the descriptions below:

```
  vec4 TextureSample(float s, float t, float r, float lodBias,
                     int texImageUnit, enum texTarget);
```

Note that not all three texture coordinates, s, t, and r, are used by all texture targets. In particular, 1D texture targets only use the s component, and 2D and rectangle (non-power-of-two) texture targets only use the s and t components. The descriptions of the texture instructions below supply all three components, as would be the case with 3D or cube map targets.

If a fragment program samples from a texture target on a texture image unit where the bound texture object is not complete, as defined in section 3.8.9, the result will be the vector (R, G, B, A) = (0, 0, 0, 1).

A fragment program will fail to load if it attempts to sample from multiple texture targets on the same texture image unit. For example, the following program would fail to load:

```
  !!ARBfp1.0
  TEX result.color, fragment.texcoord[0], texture[0], 2D;
  TEX result.depth, fragment.texcoord[1], texture[0], 3D;
  END
```

The fourth texture instruction described below, KIL, does not sample from a texture, but rather prevents further processing of the current fragment if any component of its 4-tuple vector is less than zero.

A dependent texture instruction is one that samples using a texture coordinate residing in a temporary, rather than in an attribute or a parameter. A program may have a chain of dependent texture instructions, where the result of the first texture instruction is used as the coordinate for a second texture instruction, which is in turn used as the coordinate for a third texture instruction, and so on. Each node in this chain is termed an indirection, and can be thought of as a set of texture samples that execute in parallel followed by a sequence of ALU instructions.

Some implementations may have limitations on how long the dependency chain may be, and so indirections are counted as a resource just like instructions or temporaries are counted. All programs have at least one indirection, or one node in this chain, even if the program performs no texture operation. Each instruction encountered is included in this node until a texture instruction is encountered

- whose texture coordinate is a temporary that has been previously written in the current node; or

- whose result vector is a temporary that is also the operand or result vector of a previous ALU instruction in the current node.

A new node is then started, including the texture instruction and all subsequent instructions, and the process repeats for all instructions in the program. Note that for simplicity in counting, result writemasks and operand suffixes are not taken into consideration when counting indirections.

3.11.6.1 TEX: Map coordinate to color

The TEX instruction takes the first three components of its source vector, and maps them to s, t, and r. These coordinates are used to sample from the specified texture target on the specified texture image unit in a manner consistent with its parameters. The resulting sample is mapped to RGBA as described in table 3.21 and written to the result vector.

```
  tmp = VectorLoad(op0);
  result = TextureSample(tmp.x, tmp.y, tmp.z, 0.0, op1, op2);
```

3.11.6.2 TXP: Project coordinate and map to color

The TXP instruction divides the first three components of its source vector by the fourth component and maps the results to s, t, and r. These coordinates are used to sample from the specified texture target on the specified texture image unit in a manner consistent with its parameters. The resulting sample is mapped to RGBA as described in table 3.21 and written to the result vector. If the value of the fourth component of the source vector is less than or equal to zero, the result vector is undefined.

```
  tmp = VectorLoad(op0);
  tmp.x = tmp.x / tmp.w;
  tmp.y = tmp.y / tmp.w;
  tmp.z = tmp.z / tmp.w;
  result = TextureSample(tmp.x, tmp.y, tmp.z, 0.0, op1, op2);
```

3.11.6.3 TXB: Map coordinate to color while biasing its LOD

The TXB instruction takes the first three components of its source vector and maps them to s, t, and r. These coordinates are used to sample from the specified texture target on the specified texture image unit in a manner consistent with its parameters. Additionally, the fourth component of the source vector is applied to equation 3.14 as fragment_bias below to further bias the level of detail.

```
 lambda'(x,y) = log2[p(x,y)] +
                clamp(texobj_bias + texunit_bias + fragment_bias)
```

The resulting sample is mapped to RGBA as described in table 3.21 and written to the result vector.

```
  tmp = VectorLoad(op0);
  result = TextureSample(tmp.x, tmp.y, tmp.z, tmp.w, op1, op2);
```

3.11.6.4 KIL: Kill fragment

Rather than mapping a coordinate set to a color, this function prevents a fragment from receiving any future processing. If any component of its source vector is negative, the processing of this fragment will be discontinued and no further outputs to this fragment will occur. Subsequent stages of the GL pipeline will be skipped for this fragment.

```
  tmp = VectorLoad(op0);
  if ((tmp.x < 0) || (tmp.y < 0) ||
      (tmp.z < 0) || (tmp.w < 0))
  {
      exit;
  }
```

3.11.7 Program Matrices

In addition to GL's conventional matrices, several additional program matrices are available for use as program parameters. These matrices have names of the form MATRIX<i>_ARB where <i> is between zero and <n>-1 where <n> is the value of the implementation-dependent constant MAX_PROGRAM_MATRICES_ARB. The MATRIX<i>_ARB constants obey MATRIX<i>_ARB = MATRIX0_ARB + <i>. The value of MAX_PROGRAM_MATRICES_ARB must be at least eight. The maximum stack depth for program matrices is defined by the MAX_PROGRAM_MATRIX_STACK_DEPTH_ARB and must be at least 1.

3.11.8 Required Fragment Program State

The state required to support program objects of all targets consists of:

  an integer for the program error position, initially -1;

  an array of ubytes for the program error string, initially empty;

and the state that must be maintained to indicate which integers are currently in use as program object names.

The state required to support the fragment program target consists of:

a bit indicating whether or not fragment program mode is enabled, initially disabled;

a set of MAX_PROGRAM_ENV_PARAMETERS_ARB four-component floating-point program environment parameters, initially set to (0,0,0,0);

an unsigned integer naming the currently bound fragment program, initially zero;

The state required for each fragment program object consists of:

an unsigned integer indicating the program object name;

an array of type ubyte containing the program string, initially empty;

an unsigned integer holding the length of the program string, initially zero;

an enum indicating the program string format, initially PROGRAM_FORMAT_ASCII_ARB;

a bit indicating whether or not the program exceeds the native limits;

six unsigned integers holding the number of instruction (ALU, texture, and total), texture indirection, temporary variable, and program parameter binding resources used by the program, initially all zero;

six unsigned integers holding the number of native instruction (ALU, texture, and total), texture indirection, temporary variable, and program parameter binding resources used by the program, initially all zero;

and a set of MAX_PROGRAM_LOCAL_PARAMETERS_ARB four-component floating-point program local parameters, initially set to (0,0,0,0).

Initially, no fragment program objects exist.

Additions to Chapter 4 of the OpenGL 1.3 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the OpenGL 1.3 Specification (Special Functions)

Modify Section 5.4, Display Lists (p. 191)

(modify third paragraph, p. 195) ... These are IsList, GenLists, ..., IsProgramARB, GenProgramsARB, and DeleteProgramsARB, as well as IsEnabled and all the Get commands (chapter 6).

Additions to Chapter 6 of the OpenGL 1.3 Specification (State and State Requests)

Modify Section 6.1.2, Data Conversions (p. 198)

(add before last paragraph, p. 198) The matrix selected by the current matrix mode can be queried by calling GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev with <pname> set to CURRENT_MATRIX_ARB; the matrix will be returned in transposed form with <pname> set to TRANSPOSE_CURRENT_MATRIX_ARB. The depth of the selected matrix stack can be queried with <pname> set to CURRENT_MATRIX_STACK_DEPTH_ARB. Querying CURRENT_MATRIX_ARB and CURRENT_MATRIX_STACK_DEPTH_ARB is the only means for querying the matrix and matrix stack depth of the program matrices described in section 3.11.7.

(add to end of last paragraph, p. 199) Queries of texture state variables corresponding to texture coordinate processing unit (namely, TexGen state and enables, and matrices) will produce an INVALID_OPERATION error if the value of ACTIVE_TEXTURE is greater than or equal to MAX_TEXTURE_COORDS_ARB. All other texture state queries will result in an INVALID_OPERATION error if the value of ACTIVE_TEXTURE is greater than or equal to MAX_TEXTURE_IMAGE_UNITS_ARB.

Modify Section 6.1.11, Pointer and String Queries (p. 206)

(modify last paragraph, p. 206) ... The possible values for <name> are VENDOR, RENDERER, VERSION, EXTENSIONS, and PROGRAM_ERROR_STRING_ARB.

(add after last paragraph of section, p. 207) Queries of PROGRAM_ERROR_STRING_ARB return a pointer to an implementation-dependent program load error string. If the last call to ProgramStringARB failed to load a program, the returned string describes at least one reason why the program failed to load. If the last call to ProgramStringARB successfully loaded a program, the returned string may be empty (containing only a zero terminator) or may contain one or more implementation-dependent warning messages. The contents of the error string are guaranteed to remain constant only until the next ProgramStringARB command, which may overwrite the error string.

Insert a new Section 6.1.12, Program Queries (p. 207), between existing sections 6.1.11 and 6.1.12.

6.1.12 Program Queries

The commands

```
  void GetProgramEnvParameterdvARB(enum target, uint index,
                                   double *params);
  void GetProgramEnvParameterfvARB(enum target, uint index,
                                   float *params);
```

obtain the current value for the program environment parameter numbered <index> for the given program target <target>, and places the information in the array <params>. The error INVALID_ENUM is generated if <target> specifies a nonexistent program target or a program target that does not support program environment parameters. The error INVALID_VALUE is generated if <index> is greater than or equal to the implementation-dependent number of supported program environment parameters for the program target.

When <target> is FRAGMENT_PROGRAM_ARB, each program parameter returned is an array of four values.

The commands

```
  void GetProgramLocalParameterdvARB(enum target, uint index,
                                     double *params);
  void GetProgramLocalParameterfvARB(enum target, uint index,
                                     float *params);
```

obtain the current value for the program local parameter numbered <index> belonging to the program object currently bound to <target>, and places the information in the array <params>. The error INVALID_ENUM is generated if <target> specifies a nonexistent program target or a program target that does not support program local parameters. The error INVALID_VALUE is generated if <index> is greater than or equal to the implementation-dependent number of supported program local parameters for the program target.

When the program target type is FRAGMENT_PROGRAM_ARB, each program local parameter returned is an array of four values.

The command

```
  void GetProgramivARB(enum target, enum pname, int *params);
```

obtains program state for the program target <target>, writing the state into the array given by <params>. GetProgramivARB can be used to determine the properties of the currently bound program object or implementation limits for <target>.

If <pname> is PROGRAM_LENGTH_ARB, PROGRAM_FORMAT_ARB, or PROGRAM_BINDING_ARB, GetProgramivARB returns one integer holding the program string length (in bytes), program string format, and program name, respectively, for the program object currently bound to <target>.

If <pname> is MAX_PROGRAM_LOCAL_PARAMETERS_ARB or MAX_PROGRAM_ENV_PARAMETERS_ARB, GetProgramivARB returns one integer holding the maximum number of program local parameters or program environment parameters, respectively, supported for the program target <target>.

If <pname> is MAX_PROGRAM_INSTRUCTIONS_ARB, MAX_PROGRAM_ALU_INSTRUCTIONS_ARB, MAX_PROGRAM_TEX_INSTRUCTIONS_ARB, MAX_PROGRAM_TEX_INDIRECTIONS_ARB, MAX_PROGRAM_TEMPORARIES_ARB, MAX_PROGRAM_PARAMETERS_ARB, or MAX_PROGRAM_ATTRIBS_ARB, GetProgramivARB returns a single integer giving the maximum number of total instructions, ALU instructions, texture instructions, texture indirections, temporaries, parameters, and attributes that can be used by a program of type <target>. If <pname> is PROGRAM_INSTRUCTIONS_ARB, PROGRAM_ALU_INSTRUCTIONS_ARB, PROGRAM_TEX_INSTRUCTIONS_ARB, PROGRAM_TEX_INDIRECTIONS_ARB, PROGRAM_TEMPORARIES_ARB, PROGRAM_PARAMETERS_ARB, or PROGRAM_ATTRIBS_ARB, GetProgramivARB returns a single integer giving the number of total instructions, ALU instructions, texture instructions, texture indirections, temporaries, parameters, and attributes used by the current program for <target>.

If <pname> is MAX_PROGRAM_NATIVE_INSTRUCTIONS_ARB, MAX_PROGRAM_NATIVE_ALU_INSTRUCTIONS_ARB, MAX_PROGRAM_NATIVE_TEX_INSTRUCTIONS_ARB, MAX_PROGRAM_NATIVE_TEX_INDIRECTIONS_ARB, MAX_PROGRAM_NATIVE_TEMPORARIES_ARB, MAX_PROGRAM_NATIVE_PARAMETERS_ARB, or MAX_PROGRAM_NATIVE_ATTRIBS_ARB, GetProgramivARB returns a single integer giving the maximum number of native instruction, ALU instruction, texture instruction, texture indirection, temporary, parameter, and attribute resources available to a program of type <target>. If <pname> is PROGRAM_NATIVE_INSTRUCTIONS_ARB, PROGRAM_NATIVE_ALU_INSTRUCTIONS_ARB, PROGRAM_NATIVE_TEX_INSTRUCTIONS_ARB, PROGRAM_NATIVE_TEX_INDIRECTIONS_ARB, PROGRAM_NATIVE_TEMPORARIES_ARB, PROGRAM_NATIVE_PARAMETERS_ARB, or PROGRAM_NATIVE_ATTRIBS_ARB, GetProgramivARB returns a single integer giving the number of native instruction, ALU instruction, texture instruction, texture indirection, temporary, parameter, and attribute resources consumed by the program currently bound to <target>. Native resource counts will reflect the results of implementation-dependent scheduling and optimization algorithms applied by the GL, as well as emulation of non-native features. If <pname> is PROGRAM_UNDER_NATIVE_LIMITS_ARB, GetProgramivARB returns 0 if the native resource consumption of the program currently bound to <target> exceeds the number of available resources for any resource type, and 1 otherwise.

The command

```
  void GetProgramStringARB(enum target, enum pname, void *string);
```

obtains the program string for the program object bound to <target> and places the information in the array <string>. <pname> must be PROGRAM_STRING_ARB. <n> ubytes are returned into the array program where <n> is the length of the program in ubytes, as returned by GetProgramivARB when <pname> is PROGRAM_LENGTH_ARB. The program string is always returned using the format given when the program string was specified.

The command

```
boolean IsProgramARB(uint program);
```

returns TRUE if <program> is the name of a program object. If <program> is zero or is a non-zero value that is not the name of a program object, or if an error condition occurs, IsProgramARB returns FALSE. A name returned by GenProgramsARB, but not yet bound, is not the name of a program object.

Modify Section 6.2, State Tables (p. 216)

(add to caption of Table 6.5) When accessing the current texture coordinates (CURRENT_TEXTURE_COORDS) or the texture coordinates associated with raster position (CURRENT_RASTER_TEXTURE_COORDS), the active texture unit selector (ACTIVE_TEXTURE) must be less than the implementation dependent maximum number of texture coordinate sets (MAX_TEXTURE_COORDS_ARB).

(add to caption of Table 6.8) When accessing the texture matrix stack (TEXTURE_MATRIX, TRANSPOSE_TEXTURE_MATRIX) or the texture matrix stack pointer (TEXTURE_STACK_DEPTH), the active texture unit selector (ACTIVE_TEXTURE) must be less than the implementation dependent maximum number of texture coordinate sets (MAX_TEXTURE_COORDS_ARB).

(split Table 6.17 into two tables, Texture Environment and Texture Coordinate Generation; move active texture unit selector and texture coordinate generation state to table 6.18; renumber subsequent tables)

(add to captions of Tables 6.14, 6.15, 6.16) The active texture unit selector (ACTIVE_TEXTURE) identifies which texture object is accessed, and must be less than the implementation dependent maximum number of texture image units (MAX_TEXTURE_IMAGE_UNITS_ARB).

(add to caption of Table 6.18) With the exception of ACTIVE_TEXTURE, the active texture unit selector (ACTIVE_TEXTURE) identifies which texture coordinate set is accessed, and must be less than the implementation dependent maximum number of texture coordinate sets (MAX_TEXTURE_COORDS_ARB).

Additions to Appendix A of the OpenGL 1.3 Specification (Invariance)

Add to end of Section A.3 (p. 242):

Rule 4. Fragment program instructions not relevant to the calculation of any result must have no effect on that result.

Rule 5. Fragment program instructions relevant to the calculation of any result must always produce the identical result.

Instructions relevant to the calculation of a result are any instructions in a sequence of instructions that eventually determine the source values for the calculation under consideration.

There is no guaranteed invariance between fragment colors generated by conventional GL texturing mode and fragment colors generated by fragment program mode. Multi-pass rendering algorithms that require rendering invariances to operate correctly should not mix conventional GL fragment texturing mode with fragment program mode for different rendering passes. However, such algorithms will operate correctly if the algorithms limit themselves to a single mode of fragment color generation.

There is no guaranteed invariance between the final z window coordinates of fragments processed by fragment programs that write depth values and fragments processed by any other means, even if the fragment programs in question simply copy the z value from the "fragment.position" binding. Multi-pass rendering algorithms that use depth-replacing fragment programs should use depth-replacing fragment programs on each pass to guarantee identical z values.

The texture sample chosen for a fragment of a primitive must be invariant between fragment program mode and conventional texture application mode subject to these conditions:

1. All state with the exception of fragment program state is identical

2. The primitives generating the fragments are identical

3. The sample in the fragment program mode is the result of a 'TEX' instruction (or a 'TXP' instruction with a unity q)

4. The texture coordinate operand for the texture instruction uses the same texture coordinate set as the conventional mode sample

5. The texture coordinate operand for the texture instruction has not been the result of any other operations in the fragment program

Additions to the AGL/GLX/WGL Specifications

Program objects are shared between AGL/GLX/WGL rendering contexts if and only if the rendering contexts share display lists. No change is made to the AGL/GLX/WGL API.

Changes to program objects shared between multiple rendering contexts will be serialized (i.e., the changes will occur in a specific order).

Changes to a program object made by one rendering context are not guaranteed to take effect in another rendering context until the other calls BindProgram to bind the program object.

When a program object is deleted by one rendering context, the object itself is not destroyed until it is no longer the current program object in any context. However, the name of the deleted object is removed from the program object name space, so the next attempt to bind a program using the same name will create a new program object. Recall that destroying a program object bound in

    the current rendering context effectively unbinds the object being
    destroyed.

Dependencies on OpenGL 1.4

    If OpenGL 1.4 is not supported, the modified equation for the
    calculation of level of detail by the TXB instruction in 3.11.6.3
    should read

```
      lambda'(x,y) = log2[p(x,y)] +
                     clamp(texunit_bias + fragment_bias)
```

Dependencies on EXT_vertex_weighting and ARB_vertex_blend

    If EXT_vertex_weighting and ARB_vertex_blend are both not supported,
    all discussions of multiple modelview matrices should be removed.

    In particular, the line in the grammar

```
      <stateMatrixName>      ::= "modelview" <stateOptModMatNum>
```

    should be changed to

```
      <stateMatrixName>      ::= "modelview"
```

    and the rules <stateOptModMatNum> and <stateModMatNum> should be
    deleted. The first line of Table X.2.7 should be modified to read:

```
      Binding                                Underlying State
      ------------------------------------   --------------------------
        state.matrix.modelview               modelview matrix
```

    The caption for Table X.2.7 should be modified to exclude optional
    modelview matrix number. Subsequent references to "modelview matrix
    zero" and "modelview matrix 1" should be changed to "modelview
    matrix" and the example "state.matrix.modelview[1].row[0]" should be
    changed to "state.matrix.modelview.row[0]".

Dependencies on ARB_matrix_palette:

    If ARB_matrix_palette is not supported, all discussions of the
    matrix palette should be removed.

    In particular, the line

```
      "palette" "[" <statePaletteMatNum> "]"
```

    should be removed from the <stateMatrixName> grammar rule, and the
    <statePaletteMatNum> grammar rule should be removed entirely.
    "state.matrix.palette[n]" should be removed from Table X.2.7.

Dependencies on ARB_transpose_matrix

    If ARB_transpose_matrix is not supported, the discussion of
    TRANSPOSE_CURRENT_MATRIX_ARB in the edits to section 6.1.2 should be
    removed.

Dependencies on EXT_fog_coord

```
    If EXT_fog_coord is not supported, references to "fog coordinate"
    in the definition of the "fragment.fogcoord" attribute should be
    removed.
```

Dependencies on NV_texture_rectangle

```
    If NV_texture_rectangle is not supported, the discussion of the
    rectangle (non-power-of-two) texture target in section 3.11.6 should
    be removed, and the line

      "RECT"

    should be removed from the <texTarget> grammar rule.
```

Interactions with ARB_shadow

```
    The texture comparison introduced by ARB_shadow can be expressed in
    terms of a fragment program, and in fact use the same internal
    resources on some implementations.  Therefore, if fragment program
    mode is enabled, the GL behaves as if TEXTURE_COMPARE_MODE_ARB is
    NONE.
```

Interactions with ARB_vertex_program

```
    The program object management entrypoints described in sections
    2.14.1 (for vertex programs) and 3.11.1 (for fragment programs)
    are shared by both program targets.  The PROGRAM_ERROR_STRING_ARB
    and program queries in sections 6.1.11 and 6.1.12 are also shared,
    as are all common tokens.

    The Errors section should be modified to generate INVALID_OPERATION
    from the Get command with argument CURRENT_MATRIX_ARB,
    TRANSPOSE_CURRENT_MATRIX_ARB, and CURRENT_MATRIX_STACK_DEPTH_ARB
    when the current matrix mode is TEXTURE.

    In the presence of ARB_vertex_program, ARB_fragment_program must
    recognize and return appropriate values for the GetProgram <pname>
    tokens introduced in that spec but not otherwise shared by
    ARB_fragment_program:

        PROGRAM_ADDRESS_REGISTERS_ARB                   0x88B0
        MAX_PROGRAM_ADDRESS_REGISTERS_ARB               0x88B1
        PROGRAM_NATIVE_ADDRESS_REGISTERS_ARB            0x88B2
        MAX_PROGRAM_NATIVE_ADDRESS_REGISTERS_ARB        0x88B3
```

The following tables list new program object state and implementation-dependent state:

Get Value	Type	Get Command	Initial Value	Description	Sec	Attrib
PROGRAM_ADDRESS_REGISTERS_ARB	Z+	GetProgramivARB	0	bound program address registers	6.1.12	-
PROGRAM_NATIVE_ADDRESS_ REGISTERS_ARB	Z+	GetProgramivARB	0	bound program native address registers	6.1.12	-

Table X.7. Program Object State. Program object queries return attributes of the program object currently bound to the program target <target>.

Get Value	Type	Get Command	Minimum Value	Description	Sec.	Attrib
MAX_PROGRAM_ADDRESS_REGISTERS_ARB	Z+	GetProgramivARB	0	maximum program address registers	6.1.12	-
MAX_PROGRAM_NATIVE_ADDRESS_ REGISTERS_ARB	Z+	GetProgramivARB	0	maximum program native address registers	6.1.12	-

Table X.10. New Implementation-Dependent Values Introduced by ARB_vertex_program.

In the presence of ARB_fragment_program, ARB_vertex_program must recognize and return appropriate values for the GetProgram <pname> tokens introduced in this spec. The following tables list new program object state and implementation-dependent state:

Get Value	Type	Get Command	Initial Value	Description	Sec	Attrib
PROGRAM_ALU_INSTRUCTIONS_ARB	Z+	GetProgramivARB	0	maximum program ALU instructions	6.1.12	-
PROGRAM_TEX_INSTRUCTIONS_ARB	Z+	GetProgramivARB	0	maximum program texture instructions	6.1.12	-
PROGRAM_TEX_INDIRECTIONS_ARB	Z+	GetProgramivARB	0	maximum program texture indirections	6.1.12	-
PROGRAM_NATIVE_ALU_INSTRUCTIONS_ARB	Z+	GetProgramivARB	0	maximum program native ALU instructions	6.1.12	-
PROGRAM_NATIVE_TEX_INSTRUCTIONS_ARB	Z+	GetProgramivARB	0	maximum program native texture instructions	6.1.12	-
PROGRAM_NATIVE_TEX_INDIRECTIONS_ARB	Z+	GetProgramivARB	0	maximum program native texture indirections	6.1.12	-

Table X.7. Program Object State. Program object queries return attributes of the program object currently bound to the program target <target>.

Get Value	Type	Get Command	Minimum Value	Description	Sec.	Attrib
MAX_PROGRAM_ALU_INSTRUCTIONS_ARB	Z+	GetProgramivARB	0	Number of frag. prg. ALU instructions	6.1.12	-
MAX_PROGRAM_TEX_INSTRUCTIONS_ARB	Z+	GetProgramivARB	0	Number of frag. prg. texture instructions	6.1.12	-
MAX_PROGRAM_TEX_INDIRECTIONS_ARB	Z+	GetProgramivARB	0	Number of frag. prg. texture indirections	6.1.12	-
MAX_PROGRAM_NATIVE_ALU_INSTRUCTIONS_ARB	Z+	GetProgramivARB	0	maximum program native ALU instructions	6.1.12	-
MAX_PROGRAM_NATIVE_TEX_INSTRUCTIONS_ARB	Z+	GetProgramivARB	0	maximum program native texture instructions	6.1.12	-
MAX_PROGRAM_NATIVE_TEX_INDIRECTIONS_ARB	Z+	GetProgramivARB	0	maximum program native texture indirections	6.1.12	-

Table X.10. New Implementation-Dependent Values Introduced by ARB_fragment_program.

Interactions with ATI_fragment_shader

The existing ATI_fragment_shader extension, if supported, also provides a similar fragment programming model. Mixing the two models in a single application is possible but not recommended. FRAGMENT_PROGRAM_ARB has priority over FRAGMENT_SHADER_ATI if both are enabled.

Interactions with NV_fragment_program

The NV_fragment_program extension, if supported, also provides a similar programming model. This extension is incompatible with NV_fragment_program in a number of different ways. Mixing the two models in a single application is possible but not recommended. The interactions between the extensions are defined below.

Functions, enumerants, and programs defined in NV_fragment_program are called "NV functions", "NV enumerants", and "NV programs," respectively. Functions, enumerants, and programs defined in ARB_fragment_program are called "ARB functions", "ARB enumerants", and "ARB programs," respectively.

The following GL state is identical in the two extensions:

- Fragment program mode enable. The NV and ARB enumerants have different values, but the same effect.

- Program error position.

- Program error string.

- NV_fragment_program and ARB_fragment_program "program local parameters."

- Fragment program names, targets, formats, program string, program string lengths, and residency information. The ARB and NV query functions operate differently. The ARB query function does not allow queries of target (passed in to the query) and residency information. The NV query function does not allow queries of program name (passed in to the query) or format. The format of NV programs is always PROGRAM_FORMAT_ASCII_ARB.

- Program object name space. Program objects are created differently in the NV and ARB specs. Under the NV spec, program objects are created by calling LoadProgramNV. Under the ARB spec, program objects are created by calling BindProgramARB with an unused program name.

The following state is provided only by ARB_fragment_program:

- Program environment parameters.

- Implementation-dependent limits on the number of instructions, ALU instructions, texture instructions, texture indirections, program parameters, fragment attributes, resource counts, and native resource counts. The instruction limit is baked into the NV spec. Implementations supporting NV_fragment_program have no specific restrictions on the number of ALU instructions, texture instructions, texture indirections, or fragment attributes used. Such implementations also have no limit on program parameters used, except that no more than one may be used by any single program instruction.

The following state is provided only by NV_fragment_program:

- Named program parameters (variables defined in the program text and updated by name).

The following are additional functional differences between ARB_fragment_program and NV_fragment_program:

- NV programs use a set of register names, with no support for user-defined variables (other than parameters in the program). ARB programs provide no support for fixed variable names; all variables must be declared, explicitly or implicitly, in the program.

- ARB programs support parameter variables that can be bound to selected GL state variables, and are updated automatically when the underlying state changes. NV programs provide no such support; applications must set program parameters themselves.

- ARB_fragment_program doesn't provide explicit support for multiple data types (fx12, fp16, fp32) described in NV_fragment_program, and provides no mechanism for controlling the precision used to carry out arithmetic operations.

- ARB_fragment_program doesn't support condition codes, conditional writemasks, or the "C" instruction suffix that specifies a condition code update.

- ARB_fragment_program doesn't support an absolute value operator that can be applied to a source vector as it is loaded.

- ARB_fragment_program doesn't define behavior for many floating-point special cases. On platforms where NV_fragment_program is supported, ARB programs will have the same special-case behavior.

```
    - Language to declare program parameters is slightly different
      (NV_fragment_program has "DECLARE" and "DEFINE";
      ARB_fragment_program has "PARAM").

    - NV_fragment_program provides a number of instructions not found
      in ARB_fragment_program:

        * DDX, DDY:  partial derivatives relative to x and  y.

        * "PK*" and "UP*":  packing and unpacking instructions.

        * RFL:  reflection vector.

        * SEQ, SFL, SGT, SLE, SNE, STR:  set on equal, false, greater
          than, less than or equal, not equal, and true, respectively.

        * TXD:  texture lookup w/partials.

        * X2D:  2D coordinate transformation.

    - ARB_fragment_program provides several instructions not found in
      NV_fragment_program, and there are a few instruction set
      differences:

        * ABS:  absolute value.  ABS instructions are unnecessary in
            NV_fragment_program because of the free absolute value on
            input operator.  Equivalent to:

               MOV dst, |src|;

        * CMP:  compare.  Roughly equivalent to the following
            sequence, but may be optimized further:

               SLT tmp, src0;
               LRP dst, tmp, src1, src2;

        * DPH:  homogenous dot product.  Equivalent to:

               DP3 tmp, src0, src1;
               ADD dst, tmp, src0.w;

        * KIL:  kill fragment.  Both extensions support this
            instruction, but the ARB instruction takes a vector
            operand rather than a condition code.

        * SCS:  sine/cosine.  Emulated using the separate SIN and COS
            instructions in NV_fragment_program, which also have no
            restriction on the input values.

        * SWZ:  extended swizzle.  On NV_fragment_program platforms,
            this instruction will be emulated using a single MAD
            instruction and a program parameter constant.

        * TXB:  texture sample with bias.  Not exposed in the
            NV_fragment_program API.
        * XPD:  cross product.  Emulated using a MUL and a MAD
            instruction.
```

GLX Protocol

```
    The following rendering commands are sent to the server as part of
    a glXRender request:

        BindProgramARB
            2           12              rendering command length
            2           4180            rendering command opcode
            4           ENUM            target
            4           CARD32          program

        ProgramEnvParameter4fvARB
            2           32              rendering command length
            2           4184            rendering command opcode
            4           ENUM            target
            4           CARD32          index
            4           FLOAT32         params[0]
            4           FLOAT32         params[1]
            4           FLOAT32         params[2]
            4           FLOAT32         params[3]

        ProgramEnvParameter4dvARB
            2           44              rendering command length
            2           4185            rendering command opcode
            4           ENUM            target
            4           CARD32          index
            8           FLOAT64         params[0]
            8           FLOAT64         params[1]
            8           FLOAT64         params[2]
            8           FLOAT64         params[3]

        ProgramLocalParameter4fvARB
            2           32              rendering command length
            2           4215            rendering command opcode
            4           ENUM            target
            4           CARD32          index
            4           FLOAT32         params[0]
            4           FLOAT32         params[1]
            4           FLOAT32         params[2]
            4           FLOAT32         params[3]

        ProgramLocalParameter4dvARB
            2           44              rendering command length
            2           4216            rendering command opcode
            4           ENUM            target
            4           CARD32          index
            8           FLOAT64         params[0]
            8           FLOAT64         params[1]
            8           FLOAT64         params[2]
            8           FLOAT64         params[3]

    The ProgramStringARB is potentially large, and hence can be sent in
    a glXRender or glXRenderLarge request.
    **ProgramStringARB**
        2           16+len+p        rendering command length
        2           4217            rendering command opcode
        4           ENUM            target
        4           ENUM            format
        4           sizei           len
        len         LISTofBYTE      program
        p                           unused, p=pad(len)

     If the command is encoded in a glxRenderLarge request, the
     command opcode and command length fields above are expanded to
     4 bytes each:

        4           16+len+p        rendering command length
        4           4217            rendering command opcode

The remaining commands are non-rendering commands.  These commands
are sent separately (i.e., not as part of a glXRender or
glXRenderLarge request), using the glXVendorPrivateWithReply
request:

    **DeleteProgramsARB**
        1           CARD8           opcode (X assigned)
        1           17              GLX opcode (glXVendorPrivateWithReply)
        2           4+n             request length
        4           1294            vendor specific opcode
        4           GLX_CONTEXT_TAG context tag
        4           INT32           n
        n*4         LISTofCARD32    programs

    **GenProgramsARB**
        1           CARD8           opcode (X assigned)
        1           17              GLX opcode (glXVendorPrivateWithReply)
        2           4               request length
        4           1295            vendor specific opcode
        4           GLX_CONTEXT_TAG context tag
        4           INT32           n
      =>
        1           1               reply
        1                           unused
        2           CARD16          sequence number
        4           n               reply length
        24                          unused
        n*4         LISTofCARD322   programs
        GetProgramEnvParameterfvARB
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           6               request length
            4           1296            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           ENUM            target
            4           CARD32          index
            4           ENUM            pname
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           m               reply length, m=(n==1?0:n)
            4                           unused
            4           CARD32          n (number of parameter components)

            if (n=1) this follows:

            4           FLOAT32         params
            12                          unused

            otherwise this follows:

            16                          unused
            n*4         LISTofFLOAT32   params

        GetProgramEnvParameterdvARB
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           6               request length
            4           1297            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           ENUM            target
            4           CARD32          index
            4           ENUM            pname
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           m               reply length, m=(n==1?0:n*2)
            4                           unused
            4           CARD32          n (number of parameter components)

            if (n=1) this follows:

            8           FLOAT64         params
            8                           unused

            otherwise this follows:

            16                          unused
            n*8         LISTofFLOAT64   params
```

GetProgramLocalParameterfvARB
```
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           6               request length
    4           1305            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           ENUM            target
    4           CARD32          index
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           m               reply length, m=(n==1?0:n)
    4                           unused
    4           CARD32          n (number of parameter components)

    if (n=1) this follows:

    4           FLOAT32         params
    12                          unused

    otherwise this follows:

    16                          unused
    n*4         LISTofFLOAT32   params
```

GetProgramLocalParameterdvARB
```
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           6               request length
    4           1306            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           ENUM            target
    4           CARD32          index
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           m               reply length, m=(n==1?0:n*2)
    4                           unused
    4           CARD32          n (number of parameter components)

    if (n=1) this follows:

    8           FLOAT64         params
    8                           unused

    otherwise this follows:

    16                          unused
    n*8         LISTofFLOAT64   params
```

```
GetProgramivARB
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           5               request length
    4           1307            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           ENUM            target
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           m               reply length, m=(n==1?0:n)
    4                           unused
    4           CARD32          n

    if (n=1) this follows:

    4           INT32           params
    12                          unused

    otherwise this follows:

    16                          unused
    n*4         LISTofINT32     params

GetProgramStringARB
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           5               request length
    4           1308            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           ENUM            target
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           (n+p)/4         reply length
    4                           unused
    4           CARD32          n
    16                          unused
    n           STRING          program
    p                           unused, p=pad(n)
```

```
        IsProgramARB
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           4               request length
            4           1304            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           INT32           n
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           0               reply length
            4           BOOL32          return value
            20                          unused
```

Errors

The error INVALID_OPERATION is generated by ProgramStringARB if the program string <string> is syntactically incorrect or violates any semantic restriction of the execution environment of the specified program target <target>. The error INVALID_OPERATION may also be generated by ProgramStringARB if the specified program would exceed native resource limits of the implementation.

The error INVALID_OPERATION is generated by BindProgramARB if <program> is the name of a program whose target does not match <target>.

The error INVALID_VALUE is generated by commands ProgramEnvParameter{fd}ARB, ProgramEnvParameter{fd}vARB, and GetProgramEnvParameter{fd}vARB if <index> is greater than or equal to the value of MAX_PROGRAM_ENV_PARAMETERS_ARB corresponding to the program target <target>.

The error INVALID_VALUE is generated by commands ProgramLocalParameter4{fd}ARB, ProgramLocalParameter4{fd}vARB, and GetProgramLocalParameter{fd}vARB if <index> is greater than or equal to the value of MAX_PROGRAM_LOCAL_PARAMETERS_ARB corresponding to the program target <target>.

The error INVALID_OPERATION is generated if Begin, RasterPos, or any command that performs an explicit Begin is called when fragment program mode is enabled and the currently bound fragment program object does not contain a valid fragment program.

The error INVALID_OPERATION is generated by any command accessing texture coordinate processing state if the texture unit number corresponding to the current value of ACTIVE_TEXTURE is greater than or equal to the implementation-dependent constant MAX_TEXTURE_COORDS_ARB. Such commands include: GetTexGen{if}v; TexGen{ifd}, TexGen{ifd}v; Disable, Enable, IsEnabled with argument TEXTURE_GEN_{STRQ}; Get with argument CURRENT_TEXTURE_COORDS, CURRENT_RASTER_TEXTURE_COORDS, TEXTURE_STACK_DEPTH, TEXTURE_MATRIX, TRANSPOSE_TEXTURE_MATRIX; when the current matrix mode is TEXTURE, Frustum, LoadIdentity, LoadMatrix{fd}, LoadTransposeMatrix{fd}, MultMatrix{fd}, MultTransposeMatrix{fd}, Ortho, PopMatrix, PushMatrix, Rotate{fd}, Scale{fd}, Translate{fd}.

    The error INVALID_OPERATION is generated by any command accessing
    texture image processing state if the texture unit number
    corresponding to the current value of ACTIVE_TEXTURE is greater than
    or equal to the implementation-dependent constant
    MAX_TEXTURE_IMAGE_UNITS_ARB. Such commands include: BindTexture;
    GetCompressedTexImage, GetTexEnv{if}v, GetTexImage,
    GetTexLevelParameter{if}v, GetTexParameter{if}v; TexEnv{if},
    TexEnv{if}v, TexParameter{if}, TexParameter{if}v; Disable, Enable,
    IsEnabled with argument TEXTURE_{123}D, TEXTURE_CUBE_MAP; Get with
    argument TEXTURE_BINDING_{123}D, TEXTURE_BINDING_CUBE_MAP;
    CompressedTexImage{123}D, CompressedTexSubImage{123}D,
    CopyTexImage{12}D, CopyTexSubImage{123}D, TexImage{123}D,
    TexSubImage{123}D.

New State

Get Value	Type	Get Command	Initial Value	Description	Section	Attribute
FRAGMENT_PROGRAM_ARB	B	IsEnabled	False	fragment program enable	3.8	enable
-	24+xR4	GetProgramEnv-ParameterARB	(0,0,0,0)	program environment parameters	3.11.1	-
PROGRAM_ERROR_POSITION_ARB	Z	GetIntegerv	-1	last program error position	3.11.1	-
PROGRAM_ERROR_STRING_ARB	0+xub	GetString	""	last program error string	3.11.1	-

Table X.6. New Accessible State Introduced by ARB_fragment_program.

Get Value	Type	Get Command	Initial Value	Description	Sec	Attrib
PROGRAM_BINDING_ARB	Z+	GetProgramivARB	object-specific	bound program name	6.1.12	-
PROGRAM_LENGTH_ARB	Z+	GetProgramivARB	0	bound program length	6.1.12	-
PROGRAM_FORMAT_ARB	Z1	GetProgramivARB	PROGRAM_FORMAT_ ASCII_ARB	bound program format	6.1.12	-
PROGRAM_STRING_ARB	ubxn	GetProgramStringARB	(empty)	bound program string	6.1.12	-
PROGRAM_INSTRUCTIONS_ARB	Z+	GetProgramivARB	0	bound program total instructions	6.1.12	-
PROGRAM_ALU_INSTRUCTIONS_ARB	Z+	GetProgramivARB	0	bound program ALU instructions	6.1.12	-
PROGRAM_TEX_INSTRUCTIONS_ARB	Z+	GetProgramivARB	0	bound program texture instructions	6.1.12	-
PROGRAM_TEX_INDIRECTIONS_ARB	Z+	GetProgramivARB	0	bound program texture indirections	6.1.12	-
PROGRAM_TEMPORARIES_ARB	Z+	GetProgramivARB	0	bound program temporaries	6.1.12	-
PROGRAM_PARAMETERS_ARB	Z+	GetProgramivARB	0	bound program parameter bindings	6.1.12	-
PROGRAM_ATTRIBS_ARB	Z+	GetProgramivARB	0	bound program attribute bindings	6.1.12	-
PROGRAM_NATIVE_INSTRUCTIONS_ARB	Z+	GetProgramivARB	0	bound program native instructions	6.1.12	-
PROGRAM_NATIVE_ALU_INSTRUCTIONS_ARB	Z+	GetProgramivARB	0	bound program native ALU instructions	6.1.12	-
PROGRAM_NATIVE_TEX_INSTRUCTIONS_ARB	Z+	GetProgramivARB	0	bound program native texture instructions	6.1.12	-
PROGRAM_NATIVE_TEX_INDIRECTIONS_ARB	Z+	GetProgramivARB	0	bound program native texture indirections	6.1.12	-
PROGRAM_NATIVE_TEMPORARIES_ARB	Z+	GetProgramivARB	0	bound program native temporaries	6.1.12	-
PROGRAM_NATIVE_PARAMETERS_ARB	Z+	GetProgramivARB	0	bound program native parameter bindings	6.1.12	-
PROGRAM_NATIVE_ATTRIBS_ARB	Z+	GetProgramivARB	0	bound program native attribute bindings	6.1.12	-
PROGRAM_UNDER_NATIVE_LIMITS_ARB	B	GetProgramivARB	0	bound program under native resource limits	6.1.12	-
-	24+xR4	GetProgramLocal- ParameterARB	(0,0,0,0)	bound program local parameter value	3.11.1	-

Table X.7. Program Object State. Program object queries return attributes of the program object currently bound to the program target <target>.

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
-	16+xR4	-	undefined	temporary registers	3.11.3.3	-
-	2xR4	-	undefined	fragment result registers	3.11.3.4	-

Table X.8. Fragment Program Per-fragment Execution State. All per-fragment execution state registers are uninitialized at the beginning of program execution.

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
CURRENT_MATRIX_ARB	m*n*xM^4	GetFloatv	Identity	current matrix	6.1.2	-
CURRENT_MATRIX_STACK_DEPTH_ARB	m*Z+	GetIntegerv	1	current stack depth	6.1.2	-

Table X.9. Current matrix state where m is the total number of matrices including texture matrices and program matrices and n is the number of matrices on each particular matrix stack. Note that this state is aliased with existing matrix state.

New Implementation Dependent State

Get Value	Type	Get Command	Minimum Value	Description	Sec.	Attrib
MAX_TEXTURE_COORDS_ARB	Z+	GetIntegerv	2	number of texture coordinate sets	2.7	-
MAX_TEXTURE_IMAGE_UNITS_ARB	Z+	GetIntegerv	2	number of texture image units	2.10.2	-
MAX_PROGRAM_ENV_PARAMETERS_ARB	Z+	GetProgramivARB	24	maximum program env parameters	3.11.1	-
MAX_PROGRAM_LOCAL_PARAMETERS_ARB	Z+	GetProgramivARB	24	maximum program local parameters	3.11.1	-
MAX_PROGRAM_MATRICES_ARB	Z+	GetIntegerv	8 (not to exceed 32)	maximum number of program matrices	3.11.7	-
MAX_PROGRAM_MATRIX_STACK_DEPTH_ARB	Z+	GetIntegerv	1	maximum program matrix stack depth	3.11.7	-
MAX_PROGRAM_INSTRUCTIONS_ARB	Z+	GetProgramivARB	72	maximum program total instructions	6.1.12	-
MAX_PROGRAM_ALU_INSTRUCTIONS_ARB	Z+	GetProgramivARB	48	number of frag. prg. ALU instructions	6.1.12	-
MAX_PROGRAM_TEX_INSTRUCTIONS_ARB	Z+	GetProgramivARB	24	number of frag. prg. texture instructions	6.1.12	-
MAX_PROGRAM_TEX_INDIRECTIONS_ARB	Z+	GetProgramivARB	4	number of frag. prg. texture indirections	6.1.12	-
MAX_PROGRAM_TEMPORARIES_ARB	Z+	GetProgramivARB	16	maximum program temporaries	6.1.12	-
MAX_PROGRAM_PARAMETERS_ARB	Z+	GetProgramivARB	24	maximum program parameter bindings	6.1.12	-
MAX_PROGRAM_ATTRIBS_ARB	Z+	GetProgramivARB	10	maximum program attribute bindings	6.1.12	-
MAX_PROGRAM_NATIVE_INSTRUCTIONS_ARB	Z+	GetProgramivARB	-	maximum program native total instructions	6.1.12	-
MAX_PROGRAM_NATIVE_ALU_INSTRUCTIONS_ARB	Z+	GetProgramivARB	-	maximum program native ALU instructions	6.1.12	-
MAX_PROGRAM_NATIVE_TEX_INSTRUCTIONS_ARB	Z+	GetProgramivARB	-	maximum program native texture instructions	6.1.12	-
MAX_PROGRAM_NATIVE_TEX_INDIRECTIONS_ARB	Z+	GetProgramivARB	-	maximum program native texture indirections	6.1.12	-
MAX_PROGRAM_NATIVE_TEMPORARIES_ARB	Z+	GetProgramivARB	-	maximum program native temporaries	6.1.12	-
MAX_PROGRAM_NATIVE_PARAMETERS_ARB	Z+	GetProgramivARB	-	maximum program native parameter bindings	6.1.12	-
MAX_PROGRAM_NATIVE_ATTRIBS_ARB	Z+	GetProgramivARB	-	maximum program native attribute bindings	6.1.12	-

Table X.10. New Implementation-Dependent Values Introduced by ARB_fragment_program. Values queried by GetProgram require a <pname> of FRAGMENT_PROGRAM_ARB.

Sample Usage

The following program shows how to perform a simple modulation between the interpolated color and a single texture:

```
!!ARBfp1.0
# Simple program to show how to code up the default texture environment
ATTRIB tex = fragment.texcoord;      #first set of texture coordinates
ATTRIB col = fragment.color.primary; #diffuse interpolated color
OUTPUT outColor = result.color;
TEMP tmp;
TXP tmp, tex, texture, 2D;           #sample the texture
MUL outColor, tmp, col;              #perform the modulation
END
```

The following is an example the simulates a chrome surface:

```
  !!ARBfp1.0
  ########################
  # Input Textures:
  #-----------------------
  # Texture 0 contains the default 2D texture used for general mapping
  # Texture 2 contains a 1D pointlight falloff map
  # Texture 3 contains a 2D map for calculating specular lighting
  # Texture 4 contains normalizer cube map
  # Input Texture Coordinates:
  #-----------------------
  # TexCoord1 contains the calculated normal
  # TexCoord2 contains the light to vertex vector
  # TexCoord3 contains the half-vector in tangent space
  # TexCoord4 contains the light vector in tangent space
  # TexCoord5 contains the eye vector in tangent space
  ########################
  TEMP     NdotH, lV, L;
  ALIAS    diffuse  = L;
  PARAM       half  = { 0.5, 0.5, 0.5, 0.5 };
  ATTRIB   norm_tc  = fragment.texcoord[1];
  ATTRIB     lv_tc  = fragment.texcoord[2];
  ATTRIB   half_tc  = fragment.texcoord[3];
  ATTRIB  light_tc  = fragment.texcoord[4];
  ATTRIB    eye_tc  = fragment.texcoord[5];
  OUTPUT      oCol  = result.color;
  TEX     L, light_tc, texture[4], CUBE; # Sample cube map normalizer
  # Calculate diffuse lighting (N.L)
  SUB     L, L, half;             # Bias L and then multiply by 2
  ADD     L, L, L;
  DP3     diffuse, norm_tc, L;     # N.L
  # Calculate specular lighting component { (N.H), |H|^2 }
  DP3     NdotH.x, norm_tc, half_tc;
  DP3     NdotH.y, half_tc, half_tc;
  DP3     lV.x, lv_tc, lv_tc;     # lV = (|light to vertex|)^2
  #############
  # Pass 2
  #############
  TEMP     base, specular;
  ALIAS    atten  = lV;
  TEX base, eye_tc, texture[0], 2D; # sample enviroment map using eye vector
  TEX atten,    lV,     texture[2], 1D; # Sample attenuation map
  TEX specular, NdotH,  texture[3], 2D; # Sample specular NHHH map=(N.H)^256
  # specular = (N.H)^256 * (N.L)
  # this ensures a pixel is only lit if facing the light (since the specular
  # exponent makes negative N.H positive we must do this)
  MUL      specular, specular, diffuse;
  # specular = specular * environment map
  MUL      specular, base, specular;
  # diffuse = diffuse * environment map
  MUL      diffuse, base, diffuse;
  # outColor = (specular * environment map) + (diffuse * environment map)
  ADD      base, specular, diffuse;
  # Apply point light attenutaion
  MUL      oCol, base, atten.r;
  END
```

Revision History

```
    Date: 8/22/2003
    Revision: 26
       - Added list of commands generating errors when active texture
         unit selector is out of range.
       - Fixed typo in <stateMatrixItem> rule.
       - Clarified behavior of fragment.position.z with respect to depth
         offset.

    Date: 2/26/2003
    Revision: 25
       - Fixed description of KIL instruction to reflect less than zero
         test, not less than or equal to zero.
       - Clarified the processing of incoming and outgoing depths and
         colors to reflect the conversion to floating-point on input and
         the conversion to fixed-point on output.

    Date: 1/10/2003
    Revision: 24
       - Fixed bug where "state.matrix.mvp" was specified incorrectly.
         It should be P*M0 rather than M0*P.
       - Added issue warning about CMP opcode's order of operands.

    Date: 10/22/2002
    Revision: 23
       - Fixed reference to <extSwizComp> rule in 3.11.5.28.  Instead
         reference both <xyzwExtSwizComp> and <rgbaExtSwizComp> rules.

    Date: 10/02/2002
    Revision: 22
       - Fixed typo in section 3.11.1, where 8 program environment and
         8 program local parameters are listed as the minimums instead
         of 24 of each.  Table X.10 had the correct values.
       - Fixed <stateTexEnvItem> to refer to legacy texture units.
       - Fixed typos in issue 29 pseudo-code, added some clarification.

    Date: 9/19/2002
    Revision: 21
       - Added clarifying paragraph for native texture indirection
         counting, offering examples of possible cases where texture
         indirections may be increased.
       - Fixed typos in issues 25 and 29.

    Date: 9/16/2002
    Revision: 20
       - Added precision hint program options.
       - Fixed various typos, reworded some parts for consistency.
       - Updated issues list.
```

```
Date: 9/13/2002
Revision: 19
   - Promoted minimum precision of texture coordinates in 2.1.1.
   - Added ARB_fog_* program options.
   - Removed modification to 3.9, put clamps in 3.11.4.4.
   - Made 'texture' a reserved keyword in the grammar.
   - Fixed various typos.
   - Updated section 3.11.6.
   - Updated issues list.

Date: 9/11/2002
Revision: 18
   - Updated for consistency with ARB_vertex_program revision 36.
   - Depth output moved to 3rd component of result.depth.
   - Fixed various typos, reworded things in many places.
   - Added NV_fragment_program interactions.
   - Updated issues list.

Date: 9/09/2002
Revision: 17
   - Added fogcoord and position attributes.
   - Moved fragment program section to 3.11, after fog.
   - Changed MAX_TEXTURE_UNITS/MAX_AUX_TEXTURE_UNITS to
     MAX_TEXTURE_COORDS/MAX_TEXTURE_IMAGE_UNITS.
   - Removed TRC and MOD instructions.
   - Added SIN and COS instructions.
   - Added more clarity to resource consumption wording.
   - Added invariance wording concerning depth-replacement.
   - Added rule that a program that fails to load must always fail to
     load, regardless of GL state.
   - Updated issues list.

Date: 8/30/2002
Revision: 16
   - Improved texture indirection description.
   - Defined result of sample from incomplete texture as (0,0,0,1).
   - Removed PROGRAMS_LOAD_OVER_NATIVE_LIMITS_ARB per-target query.
   - Allowed ProgramStringARB to fail on non-native programs.
   - Updated issues list.

Date: 8/28/2002
Revision: 15
   - Updated for consistency with ARB_vertex_program revision 35.
   - Added PROGRAMS_LOAD_OVER_NATIVE_LIMITS_ARB per-target query.
   - Changed MAX_AUX_TEXTURE_UNITS_ARB enum value.
   - Updated issues list.

Date: 8/22/2002
Revision: 14
   - Added sine/cosine instruction (SCS).
   - Updated texture sample grammar, replaced texenables hierarchy.
   - Added EXT_vertex_weighting and ARB_vertex_blend dependency.
   - Updated issues list.
```

```
Date: 8/14/2002
Revision: 13
   - Fixed <paramConstant> grammar rule.
   - Updated issues list.

Date: 8/06/2002
Revision: 12
   - Fixed various typos.
   - Updated issues list.
   - Added wording to 3.10.3.6 to reflect that native resource
     consumption may increase due to emulated instructions.

Date: 7/29/2002
Revision: 11
   - Updated for consistency with ARB_vertex_program revision 34.
   - Added support for matrix binding.
   - Removed precision queries.
   - Updated issues list.

Date: 7/16/2002
Revision: 10
   - Updated for consistency with ARB_vertex_program revision 31.
   - Added fog params and depth range bindings to grammar.
   - Removed stpq writemasks and swizzles from grammar.
   - Required swizzle components to come from same set, xyzw or rgba.

Date: 7/10/2002
Revision: 9
   - Made fog params and depth range bindable.
   - Changed texture instruction names to match 3-letter format.
   - Made texture instructions more consistent with ALU instructions.
   - Increased minimums for implementation-dependent values.
   - Re-introduced 4-components swizzles and the SWZ instruction.
   - Updated issues list.

Date: 7/03/2002
Revision: 8
   - Fixed typos.
   - Added DST, LIT, SGE, SLT instructions.
   - Changed FRC definition to match ARB_vertex_program, added MOD
     instruction to expose fmod(arg, 1.0) behavior.

Date: 6/25/2002
Revision: 7
   - Updated for consistency with ARB_vertex_program revision 29.

Date: 6/19/2002
Revision: 6
   - Updated for consistency with ARB_vertex_program revision 28.
   - Changed from ATI to ARB prefix/suffix.
   - Started using single integer revision number.
   - Added a few more issues to the list.

Date: 6/14/2002
Revision: 1.4
   - Updated for consistency with ARB_vertex_program revision 27.
   - Added a few more issues to the list.
```

```
    Date: 6/05/2002
    Revision: 1.3
       - Updated for consistency with ARB_vertex_program revision 26.
       - Incorporated program object management, removing dependency on
         ARB_vertex_program.
       - Added interaction with ARB_shadow.

    Date: 6/03/2002
    Revision: 1.2
       - Updated for consistency with ARB_vertex_program revision 25.
       - Fixed TexInstructions to use <texSrcReg>, i.e. no parameters.
       - Added TRC, POW, DPH instructions, updated FRC and LRP.
       - Added fog color parameter binding.

    Date: 5/23/2002
    Revision: 1.1
       - Updated for consistency with ARB_vertex_program revision 24.
       - Added GetProgramfvATI entrypoint for querying precision values.

    Date: 5/10/2002
    Revision: 1.0
       - First draft for circulation.
```

Name

    ARB_fragment_program_shadow

Name Strings

    GL_ARB_fragment_program_shadow

IP Status

    Unknown, but Microsoft claims to own intellectual property
    related to ARB_fragment_program. This extension is
    an extension to ARB_fragment_program.

Status

    Complete. Approved by ARB on December 16, 2003

Version

    Last Modified Date: December 8, 2003
    Revision: 5

Number

    ARB Extension #36

Dependencies

    The extension is written against the OpenGL 1.3 Specification.

    ARB_fragment_program is required.

    ARB_shadow is required.

    EXT_texture_rectange affects the definition of this extension.

Overview

    This extension extends ARB_fragment_program to remove
    the interaction with ARB_shadow.

    This extension defines the program option
    "ARB_fragment_program_shadow".

    If a fragment program specifies the option
    "ARB_fragment_program_shadow"

        SHADOW1D, SHADOW2D, SHADOWRECT

    are added as texture targets. When shadow map comparisons are
    desired, specify the SHADOW1D, SHADOW2D, or SHADOWRECT texture
    targets in texture instructions.

    Programs must assure that the comparison mode for each depth
    texture (TEXTURE_COMPARE_MODE) and/or the internal texture
    format (DEPTH_COMPONENT) and the targets of the texture lookup

instructions match. Otherwise, if the comparison mode
and/or the internal texture format are inconsistent with the
texture target, the results of the texture lookup are undefined.

Issues

(1) What should this extension be called?

RESOLVED: ARB_fragment_program_shadow. Shadow support
is the only new feature. The name ARB_fragment_program2
should be used for a far more major revision to
ARB_fragment_program. ARB_fragment_program1_1 is
less descriptive.

*(2) Should this extension use the header string "!!ARBfp1.1" or
a program option "ARB_fragment_program_shadow"?*

RESOLVED: Program option "ARB_fragment_program_shadow".

(3) What form should the ARB_fragment_program_shadow option take?

a. New sampler instructions.
```
SHX result.color.a, fragment.texcoord[1], texture[0], 2D;
```

b. New texture modifiers.
```
TEX result.color.a, fragment.texcoord[1], texture[0], 2D,SHADOW;
```

c. New texture targets.
```
TEX result.color.a, fragment.texcoord[1], texture[0], SHADOW2D;
```

d. New sampler instructions AND new texture modifiers.
```
SHX result.color.a, fragment.texcoord[1], texture[0], 2D,SHADOW;
```

e. New sampler instructions AND new texture targets.
```
SHX result.color.a, fragment.texcoord[1], texture[0], SHADOW2D;
```

RESOLVED: Choose the simplest option c, add new texture targets.

All of the above forms are functionally equivalent.

An earlier draft proposed option a, adding six new shadow
instructions. The required shadow instructions are
three variants of shadow instruction (non-projective, projective,
and biased), and the same instructions with the modifier _SAT.

Option b adds texture modifiers but requires additional semantic
restrictions.

Option c adds texture targets only. It is a sufficient
and simple change to one grammar rule.

Option d and e are listed for completeness. They require
additional instructions and additional semantic restrictions.

Note that option e is most similar to the resolution of this issue
by ARB_fragment_shader and the OpenGL Shading Language. The OpenGL
Shading Language has both built-in texture and shadow functions and

sampler types, analogous to texture instructions and texture targets. The resolution here drops the added reduntancy and potential error checking in favor of simplicity, but is otherwise consistent. This resolution is also consistent with the precident already established in ARB_fragment_program, since we have a TEX instruction, not a TEX1D, TEX2D, TEXCUBE, TEX3D, TEXRECT instructions.

(4) How should ARB_fragment_program_shadow function?

a. Simply remove the interaction with ARB_shadow so that TEXTURE_COMPARE_MODE behaves exactly as specified in the OpenGL 1.4 specification.

b. Add "SHADOW" targets to texture lookup instructions. TEXTURE_COMPARE_MODE is ignored. For samples from a SHADOW target TEXTURE_COMPARE_MODE is treated as COMPARE_R_TO_TEXTURE; otherwise, it is treated as NONE.

c. Like (b), but with undefined results if TEXTURE_COMPARE_MODE and/or the internal format of the texture does not match the target.

d. A hybrid of (a) and (b), where the SHADOW target means to use the TEXTURE_COMPARE_MODE state.

RESOLVED - Option c, undefined behavior when the target and mode do not match.

Program text is not simply loaded, it is compiled, optimized and then loaded. Options a and d would remove information from the optimizer. Which components of the texture coordinate are required for the sample? Specifically, is the r component of the texture coordinate required? Options b and c are both sufficient and retain the information required by optimizers. Option c is consistent with the resolution chosen by ARB_fragment_shader.

(5) What if additional texture compare modes are added by future extensions to ARB_SHADOW?

We do not anticipate future extensions adding additional texture compare modes. Only the additional mode COMPARE_T_TO_TEXTURE has even marginal utility, and then only for SHADOW1D targets. However, a future extension adding additional texture compare modes is not precluded. The language in this specification is carefully, if somewhat awkwardly, written to say the TEXTURE_COMPARE_MODE either "is NONE" or "is not NONE.

(6) Does EXT_shadow_funcs interact with this extension?

RESOLVED: It doesn't. ARB_shadow supports LEQUAL or GEQUAL comparison functions. EXT_shadow_funcs simply adds the additional functions LESS, GREATER, EQUAL, NOTEQUAL, ALWAYS, and NEVER. Whichever function is specified will be used for the comparison function.

(7) Does ARB_shadow_ambient interact with this extension?

RESOLVED: It doesn't. ARB_shadow returns a result in the range [0,1]. ARB_shadow_ambient simply maps this range to [TEXTURE_COMPARE_FAIL_ARB, 1]. The result will be returned in the specified range.

(8) How would an existing fragment program be ported to use the program option ARB_fragment_program_shadow?

RESOLVED: Fairly simply, but with a caveat on undefined behavior.

```
!!ARBfp1.0
# A simple example of shadow map (R <= Dt)
#
# SHOULD make sure that the 2D texture bound to texture unit 0:
#    texture format of DEPTH_COMPONENT (for highest quality comparison)
#    TEXTURE_MAG_FILTER is NEAREST
#    TEXTURE_MIN_FILTER is NEAREST or NEAREST_MIPMAP_NEAREST
# Assumes DEPTH_TEXTURE_MODE is LUMINANCE or INTENSITY
#
TEMP Result;
ALIAS Dt = Result;
TEX Dt, fragment.texcoord[0], texture[0], 2D;
SGE Result, Dt.x, fragment.texcoord[0].z;      # R <= Dt

!!ARBfp1.0
OPTION ARB_fragment_program_shadow;
# A simple example of shadow map (R<= Dt)
#
# MUST make sure that the 2D texture bound to texture unit 0:
#    texture format of DEPTH_COMPONENT and a
#    TEXTURE_COMPARE_MODE of COMPARE_R_TO_TEXTURE
# Otherwise, the Result is undefined.
#
# Remember also that to get R <= Dt to set:
#    TEXTURE_COMPARE_FUNC of LEQUAL
#
# A single compare equivalent to the above example will result if:
#    TEXTURE_MAG_FILTER is NEAREST
#    TEXTURE_MIN_FILTER is NEAREST or NEAREST_MIPMAP_NEAREST
# Otherwise, percent closer filtering may be applied.
#
TEMP Result;
TEX Result, fragment.texcoord[0], texture[0], SHADOW2D;
```

New Procedures and Functions

None

New Tokens

None

Additions to Chapter 2 of the OpenGL 1.3 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the OpenGL 1.3 Specification (Rasterization)

Modify Section 3.11.2 Fragment Program Grammar and Semantic Restrictions

Replace <texTarget> grammar rule with

```
<texTarget>             ::= "1D"
                          | "2D"
                          | "3D"
                          | "CUBE"
                          | "RECT"
                          | <shadowTarget> (if program option is present)

<shadowTarget>          ::= "SHADOW1D"
                          | "SHADOW2D"
                          | "SHADOWRECT"
```

Add Section 3.11.4.5.3 Fragment Program Shadow Option

If a fragment program specifies the "ARB_fragment_program_shadow" program option, the <texTarget> rule is modified to add the texture targets SHADOW1D, SHADOW2D and SHADOWRECT (See Section 3.11.2).

Modify Section 3.11.6 Fragment Program Texture Instruction Set

(replace 1st through 4th paragraphs with the following paragraphs)

The first three texture instructions described below specify the mapping of 4-tuple input vectors to 4-tuple output vectors. The sampling of the texture works as described in section 3.8, except that texture environments and texture functions are not applicable, and the texture enables hierarchy is replaced by explicit references to the desired texture target (i.e., 1D, 2D, 3D, cube map, rectangle). These texture instructions specify how the 4-tuple is mapped into the coordinates used for sampling. The following function is used to describe the texture sampling in the descriptions below:

```
  vec4 TextureSample(float s, float t, float r, float lodBias,
                     int texImageUnit, enum texTarget);
```

Note that not all three texture coordinates, s, t, and r, are used by all texture targets. In particular, 1D texture targets only use the s component. 2D and RECT (non-power-of-two) texture targets only use the s and t components. SHADOW1D texture targets only use the s and r components. The descriptions of the texture instructions below supply all three components, as would be the case with CUBE, 3D, SHADOW2D, and SHADOWRECT targets.

If a fragment program samples from a texture target on a texture image unit where the bound texture object is not complete, as defined in section 3.8.9, the result will be the vector (R, G, B, A) = (0, 0, 0, 1).

If a fragment program does not specify the "ARB_fragment_program_shadow" program option, and if a fragment

    program samples from a texture target of 1D, 2D, or RECT, it is as
    if TEXTURE_COMPARE_MODE_ARB is NONE.

    If a fragment program specifies the "ARB_fragment_program_shadow"
    program option, the result returned of a sample from a texture target
    on a texture image unit is undefined if:

```
      the texture target is 1D, 2D, or RECT, and
      the texture object's internal format is DEPTH_COMPONENT_ARB, and
      the TEXTURE_COMPARE_MODE_ARB is not NONE;
```

    or

```
      the texture target is SHADOW1D, SHADOW2D, SHADOWRECT, and
        the texture object's internal format is DEPTH_COMPONENT_ARB, and
      the TEXTURE_COMPARE_MODE_ARB is NONE;
```

    or

```
      the texture target is SHADOW1D, SHADOW2D, SHADOWRECT, and
        the texture object's internal format is not DEPTH_COMPONENT_ARB.
```

    A fragment program will fail to load if it attempts to sample from
    multiple texture targets on the same texture image unit. For example,
    the following programs would fail to load:

```
      !!ARBfp1.0
      TEX result.color.rgb, fragment.texcoord[0], texture[0], 2D;
      TEX result.color.a,   fragment.texcoord[1], texture[0], 3D;
      END

      !!ARBfp1.0
      OPTION ARB_fragment_program_shadow;
      TEX result.color.rgb, fragment.texcoord[0], texture[0], 2D;
      TEX result.color.a,   fragment.texcoord[1], texture[0], SHADOW2D;
      END
```

Additions to Chapter 4 of the OpenGL 1.3 Specification (Per-Fragment Operations and the Frame Buffer)

    None

Additions to Chapter 5 of the OpenGL 1.3 Specification (Special Functions)

    None

Additions to Chapter 6 of the OpenGL 1.3 Specification (State and State Requests)

    None

Additions to Appendix A of the OpenGL 1.3 Specification (Invariance)

    None

Additions to the AGL/GLX/WGL Specifications

    None

Dependencies on EXT_texture_rectangle

    If EXT_texture_rectangle is not supported:

    Section 3.11.2 should be modified by removing the line:

```
      | "SHADOWRECT"
```

    from the <shadowTarget> grammar rule;

    and Section 3.11.6 should be modified by removing the discussion
    of the rectangle shadow texture target.

Name

    ARB_half_float_pixel

Name Strings

    GL_ARB_half_float_pixel

Contributors

    Pat Brown
    Jon Leech
    Rob Mace
    Brian Paul

Contact

    Dale Kirkland, NVIDIA (dkirkland 'at' nvidia.com)

Status

    Complete. Appprove by the ARB on October 22, 2004.

Version

    Last Modified Date: October 1, 2004
    Version: 6

Number

    ARB Extension #40

Dependencies

    This extension is written against the OpenGL 2.0 Specification
    but will work with the OpenGL 1.5 Specification.

    Based on the NV_half_float extension.

    This extension interacts with ARB_color_buffer_float.

Overview

    This extension introduces a new data type for half-precision (16-bit)
    floating-point quantities. The floating-point format is very similar
    to the IEEE single-precision floating-point standard, except that it
    has only 5 exponent bits and 10 mantissa bits. Half-precision floats
    are smaller than full precision floats and provide a larger dynamic
    range than similarly sized normalized scalar data types.

    This extension allows applications to use half-precision floating-
    point data when specifying pixel data. It extends the existing image
    specification commands to accept the new data type.

    Floating-point data is clamped to [0, 1] at various places in the
    GL unless clamping is disabled with the ARB_color_buffer_float
    extension.

IP Status

SGI owns US Patent #6,650,327, issued November 18, 2003. SGI believes this patent contains necessary IP for graphics systems implementing floating point (FP) rasterization and FP framebuffer capabilities.

SGI will not grant the ARB royalty-free use of this IP for use in OpenGL, but will discuss licensing on RAND terms, on an individual basis with companies wishing to use this IP in the context of conformant OpenGL implementations. SGI does not plan to make any special exemption for open source implementations.

Contact Doug Crisman at SGI Legal for the complete IP disclosure.

Issues

1. *How is this extension different from the NV_half_float extension?*

   This extension does not add new commands for specifying half-precision vertex data, and all imaging functions have been listed for supporting the "half" type.

2. *What should the new data type be called? "half"? "hfloat"?*

   RESOLVED: half . This convention builds on the convention of using the type "double" to describe double-precision floating-point numbers. Here, "half" will refer to half-precision floating-point numbers.

   Even though the 16-bit float data type is a first-class data type, it is still more problematic than the other types in the sense that no native programming languages support the data type. "hfloat/hf" would have reflected a second-class status better than "half/h".

   Both names are not without conflicting precedents. The name "half" is used to connote 16-bit scalar values on some 32-bit CPU architectures (e.g., PowerPC). The name "hfloat" has been used to describe 128-bit floating-point data on VAX systems.

3. *Should half-precision data be accepted by commands in the imaging subset that accept pixel data?*

   RESOLVED: Yes, all functions in the core OpenGL and the imaging subset that accept pixel data accept half-precision data.

4. *Should the special representations NaN, INF, and denormal be supported?*

   RESOLVED: Implementation dependent. The spec reflects that Nan and INF produce unspecified results. Denormalized numbers can be treated as a value of 0.

New Tokens

    Accepted by the <type> parameter of DrawPixels, ReadPixels,
    TexImage1D, TexImage2D, TexImage3D, GetTexImage, TexSubImage1D,
    TexSubImage2D, TexSubImage3D, GetHistogram, GetMinmax,
    ConvolutionFilter1D, ConvolutionFilter2D, GetConvolutionFilter,
    SeparableFilter2D, GetSeparableFilter, ColorTable, ColorSubTable,
    and GetColorTable:

        HALF_FLOAT_ARB   0x140B

Additions to Chapter 2 of the OpenGL 2.0 Specification (OpenGL Operation)

Add a new Section 2.1.2, (p. 6):

2.1.2 16-Bit Floating-Point Numbers

A 16-bit floating-point number has a 1-bit sign (S), a 5-bit exponent (E), and a 10-bit mantissa (M). The value of a 16-bit floating-point number is determined by the following:

```
    (-1)^S * 0.0,                        if E == 0 and M == 0,
    (-1)^S * 2^-14 * (M / 2^10),         if E == 0 and M != 0,
    (-1)^S * 2^(E-15) * (1 + M/2^10),    if 0 < E < 31,
    (-1)^S * INF,                        if E == 31 and M == 0, or
    NaN,                                 if E == 31 and M != 0,
```

where

```
    S = floor((N mod 65536) / 32768),
    E = floor((N mod 32768) / 1024), and
    M = N mod 1024.
```

Implementations are also allowed to use any of the following alternative encodings:

```
    (-1)^S * 0.0,                        if E == 0 and M != 0,
    (-1)^S * 2^(E-15) * (1 + M/2^10),    if E == 31 and M == 0, or
    (-1)^S * 2^(E-15) * (1 + M/2^10),    if E == 31 and M != 0,
```

Any representable 16-bit floating-point value is legal as input to a GL command that accepts 16-bit floating-point data. The result of providing a value that is not a floating-point number (such as infinity or NaN) to such a command is unspecified, but must not lead to GL interruption or termination. Providing a denormalized number or negative zero to GL must yield predictable results.

(modify Table 2.2, p. 9) -- add new row

GL Type	Minimum Bit Width	Description
half	16	half-precision floating-point value encoded in an unsigned scalar

Modify Section 2.14, (Colors and Coloring), p. 59

(modify Table 2.9, p. 59) Add new row to the table:

GL Type	Conversion
half	c

Additions to Chapter 3 of the OpenGL 2.0 Specification (Rasterization)

Modify Section 3.6.4, Rasterization of Pixel Rectangles (p. 126)

(modify Table 3.5, p. 128 -- add new row)

type Parameter Token Name	Corresponding GL Data Type	Special Interpretation
HALF_FLOAT_ARB	half	No

(modify Unpacking, p. 129) Data are taken from host memory as a sequence of signed or unsigned bytes (GL data types byte and ubyte), signed or unsigned integers (GL data types int and uint), or floating-point values (GL data types half and float).

Additions to Chapter 4 of the OpenGL 2.0 Specification (Per-Fragment Operations and the Framebuffer)

Modify Section 4.3.2, Reading Pixels (p. 219)

(modify Final Conversion, p. 222) For an index, if the type is not FLOAT or HALF_FLOAT_ARB, final conversion consists of masking the index with the value given in Table 4.6; if the type is FLOAT or HALF_FLOAT_ARB, then the integer index is converted to a GL float or half data value.

(modify Table 4.7, p. 224 -- add new row)

type Parameter	GL Data Type	Component Conversion Formula
HALF_FLOAT_ARB	half	c = f

Additions to Chapter 5 of the OpenGL 2.0 Specification (Special Functions)

None

Additions to Chapter 6 of the OpenGL 2.0 Specification (State and State Requests)

None

Additions to the AGL/GLX/WGL Specifications

None

GLX Protocol (Modification to the GLX 1.3 Protocol Encoding Specification)

Modify Appendix A, Pixel Data (p. 148)

(Modify Table A.1, p. 149 -- add new row for HALF_FLOAT_ARB data)

```
    type             Encoding    Protocol Type    nbytes
    -------------    --------    -------------    ------
    HALF_FLOAT_ARB    0x140B      CARD16           2
```

Errors

None

New State

None

New Implementation Dependent State

None

Revision History

```
Rev.    Date    Author    Changes
----  -------- ---------  ------------------------------------------
  1   12/15/03 Kirkland   Initial version based on the NV_half_float
                          specification.

  2   2/26/04  Kirkland   Changed NVIDIA_xxx to NV_xxx.
                          Changed the issue resolution for INF and NaN.

  3   3/11/04  Kirkland   Updated language for float16 number handling.
                          Added bit encodings for half values.
                          Added issue dealing with name "half".

  4   7/23/04  Kirkland   Added alternative encodings options for
                          float16 format.

  5   9/17/04  Kirkland   Updated to reference OpenGL 2.0 spec.

  6   10/1/04  Kirkland   Updated IP section.
```

Name

    ARB_imaging

Name Strings

    GL_ARB_imaging

NOTE: This extension does not have its own specification document, since it has been included in the OpenGL 1.2.1 Specification (downloadable from www.opengl.org). Please refer to the 1.2.1 Specification for more information.

Name

    ARB_multisample

Name Strings

    GL_ARB_multisample
    GLX_ARB_multisample
    WGL_ARB_multisample

Status

    Approved by ARB on 12/8/1999.
    GLX protocol must still be defined.

Version

    Last Modified Date: December 15, 1999
    Author Revision: 0.5

    Based on: SGIS_Multisample Specification
               Date: 1994/11/22 Revision: 1.14

Number

    ARB Extension #5

Dependencies

    WGL_EXT_extensions_string is required.
    WGL_EXT_pixel_format is required.

Overview

This extension provides a mechanism to antialias all GL primitives: points, lines, polygons, bitmaps, and images. The technique is to sample all primitives multiple times at each pixel. The color sample values are resolved to a single, displayable color each time a pixel is updated, so the antialiasing appears to be automatic at the application level. Because each sample includes depth and stencil information, the depth and stencil functions perform equivalently to the single-sample mode.

An additional buffer, called the multisample buffer, is added to the framebuffer. Pixel sample values, including color, depth, and stencil values, are stored in this buffer. When the framebuffer includes a multisample buffer, it does not also include separate depth or stencil buffers, even if the multisample buffer does not store depth or stencil values. Color buffers (left/right, front/ back, and aux) do coexist with the multisample buffer, however.

Multisample antialiasing is most valuable for rendering polygons, because it requires no sorting for hidden surface elimination, and it correctly handles adjacent polygons, object silhouettes, and even intersecting polygons. If only points or lines are being rendered, the "smooth" antialiasing mechanism provided by the base GL may result in a higher quality image. This extension is

designed to allow multisample and smooth antialiasing techniques
to be alternated during the rendering of a single scene.

IP Status

TBD

Issues

1. Multiple passes have been taken out. Is this acceptable?

   RESOLUTION: Yes. This can be added back with an additional
   extension if needed.

2. Would SampleAlphaARB be a better name for the function
   SampleMaskARB? If so, the name SAMPLE_MASK_ARB should also be
   changed to SAMPLE_ALPHA_ARB.

   RESOLUTION: Names containing "mask" were changed to use
   "coverage" instead.

3. Should the SampleCoverageARB function be changed to allow
   blending between more than two objects?

   RESOLUTION: Not addressed by this extension. An additional
   extension has been proposed that allows a coverage range for
   each object. The coverage range is a min and max value that
   can be used to blend multiple objects at different level-of-
   detail fading. The SampleCoverageARB function will layer on
   this new extension.

New Procedures and Functions

```
void SampleCoverageARB(clampf value,
                       boolean invert);
```

New Tokens

Accepted by the <attribList> parameter of glXChooseVisual, and by
the <attrib> parameter of glXGetConfig:

```
GLX_SAMPLE_BUFFERS_ARB              100000
GLX_SAMPLES_ARB                     100001
```

Accepted by the <piAttributes> parameter of
wglGetPixelFormatAttribivEXT, wglGetPixelFormatAttribfvEXT, and
the <piAttribIList> and <pfAttribIList> of wglChoosePixelFormatEXT:

```
WGL_SAMPLE_BUFFERS_ARB              0x2041
WGL_SAMPLES_ARB                     0x2042
```

    Accepted by the <cap> parameter of Enable, Disable, and IsEnabled,
    and by the <pname> parameter of GetBooleanv, GetIntegerv,
    GetFloatv, and GetDoublev:

MULTISAMPLE_ARB	0x809D
SAMPLE_ALPHA_TO_COVERAGE_ARB	0x809E
SAMPLE_ALPHA_TO_ONE_ARB	0x809F
SAMPLE_COVERAGE_ARB	0x80A0

    Accepted by the <mask> parameter of PushAttrib:

MULTISAMPLE_BIT_ARB	0x20000000

    Accepted by the <pname> parameter of GetBooleanv, GetDoublev,
    GetIntegerv, and GetFloatv:

SAMPLE_BUFFERS_ARB	0x80A8
SAMPLES_ARB	0x80A9
SAMPLE_COVERAGE_VALUE_ARB	0x80AA
SAMPLE_COVERAGE_INVERT_ARB	0x80AB

Additions to Chapter 2 of the 1.2.1 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.2.1 Specification (Rasterization)

If SAMPLE_BUFFERS_ARB is a value of one, the rasterization of all GL primitives is changed, and is referred to as multisample rasterization. Otherwise, primitive rasterization operates as it is described in the GL specification, and is referred to as single-sample rasterization. The value of SAMPLE_BUFFERS_ARB is an implementation dependent constant, and is queried by calling GetIntegerv with <pname> set to SAMPLE_BUFFERS_ARB. This value is the same as GLX_SAMPLE_BUFFERS_ARB or WGL_SAMPLE_BUFFERS_ARB for the visual or pixel format associated with the context.

During multisample rendering the contents of a pixel fragment are changed in two ways. First, each fragment includes a coverage value with SAMPLES_ARB bits. The value of SAMPLES_ARB is an implementation-dependent constant, and is queried by calling GetIntegerv with <pname> set to SAMPLES_ARB. Second, each fragment includes SAMPLES_ARB depth values, instead of the single depth value that is maintained in single-sample rendering mode. Each pixel fragment thus consists of integer x and y grid coordinates, a color, SAMPLES_ARB depth values, texture coordinates, and a coverage value with a maximum of SAMPLES_ARB bits.

The behavior of multisample rasterization is a function of MULTISAMPLE_ARB, which is enabled and disabled by calling Enable or Disable, with <cap> set to MULTISAMPLE_ARB. Its value is queried using IsEnabled, with <cap> set to MULTISAMPLE_ARB.

If MULTISAMPLE_ARB is disabled, multisample rasterization of all primitives is equivalent to single-sample rasterization, except that the fragment coverage value is set to full coverage. The depth values may all be set to the single value that would have been assigned by single-sample rasterization, or they may be assigned as described below for multisample rasterization.

If MULTISAMPLE_ARB is enabled, multisample rasterization of all primitives differs substantially from single-sample rasterization. It is understood that each pixel in the framebuffer has SAMPLES_ARB locations associated with it. These locations are exact positions, rather than regions or areas, and each is referred to as a sample point. The sample points associated with a pixel may be located inside or outside of the unit square that is considered to bound the pixel. Furthermore, the relative locations of sample points may be identical for each pixel in the framebuffer, or they may differ.

If the sample locations differ per pixel, they should be aligned to window, not screen, boundaries. Otherwise rendering results will be window-position specific. The invariance requirement described in section 3.1 is relaxed for all enabled multisample rendering, because the sample locations may be a function of pixel location.

It is not possible to query the actual sample locations of a pixel.

Point Multisample Rasterization
[Insert before section 3.3.1]

If MULTISAMPLE_ARB is enabled, and SAMPLE_BUFFERS_ARB is a value of one, then points are rasterized using the following algorithm, regardless of whether point antialiasing (POINT_SMOOTH) is enabled or disabled. Point rasterization produces a fragment for each framebuffer pixel with one or more sample points that intersect the region lying within the circle having diameter equal to the current point width and centered at the point's (Xw,Yw). Coverage bits that correspond to sample points that intersect the circular region are 1, other coverage bits are 0. All depth values of the fragment are assigned the depth value of the point being rasterized. Other data associated with each fragment are the data associated with the point being rasterized.

Point size range and number of gradations are equivalent to those supported for antialiased points.

Line Multisample Rasterization
[Insert before section 3.4.3]

If MULTISAMPLE_ARB is enabled, and SAMPLE_BUFFERS_ARB is a value of one, then lines are rasterized using the following algorithm, regardless of whether line antialiasing (LINE_SMOOTH) is enabled or disabled. Line rasterization produces a fragment for each framebuffer pixel with one or more sample points that intersect the rectangular region that is described in the Antialiasing section of 3.4.2 (Other Line Segment Features). If line stippling is enabled, the rectangular region is subdivided into adjacent unit-length rectangles, with some rectangles eliminated according to the procedure given under Line Stipple, where "fragment" is replaced by "rectangle".

Coverage bits that correspond to sample points that intersect a retained rectangle are 1, other coverage bits are 0. Each depth value is produced by substituting the corresponding sample location into equation 3.1, then using the result to evaluate equation 3.3. The data associated with each fragment are otherwise computed by evaluating equation 3.1 at the fragment center, then substituting into equation 3.2.

Line width range and number of gradations are equivalent to those supported for antialiased lines.

Polygon Multisample Rasterization
[Insert before section 3.5.6]

If MULTISAMPLE_ARB is enabled, and SAMPLE_BUFFERS_ARB is a value of one, then polygons are rasterized using the following algorithm, regardless of whether polygon antialiasing (POLYGON_SMOOTH) is enabled or disabled. Polygon rasterization produces a fragment for each framebuffer pixel with one or more sample points that satisfy the point sampling criteria described in section 3.5.1, including the special treatment for sample points that lie on a polygon boundary edge. If a polygon is culled, based on its orientation and the CullFace mode, then no fragments are produced during rasterization. Fragments are culled by the polygon stipple just as they are for aliased and antialiased polygons.

Coverage bits that correspond to sample points that satisfy the point sampling criteria are 1, other coverage bits are 0. Each depth value is produced by substituting the corresponding sample location into the barycentric equations described in section 3.5.1, using the approximation to equation 3.4 that omits w components. The data associated with each fragment are otherwise computed by barycentric evaluation using the fragment's center point.

The rasterization described above applies only to the FILL state of PolygonMode. For POINT and LINE, the rasterizations described in the Point Multisample Rasterization and the Line Multisample Rasterization sections apply.

Pixel Rectangle Multisample Rasterization
[Insert before section 3.6.5]

If MULTISAMPLE_ARB is enabled, and SAMPLE_BUFFERS_ARB is a value of one, then pixel rectangles are rasterized using the following algorithm. Let (Xrp,Yrp) be the current raster position. (If the current raster position is invalid, then DrawPixels is ignored.) If a particular group (index or components) is the nth in a row and belongs to the mth row, consider the region in window coordinates bounded by the rectangle with corners

```
  (Xrp + Zx*n, Yrp + Zy*m)
```

and

```
  (Xrp + Zx*(n+1), Yrp + Zy*(m+1))
```

where Zx and Zy are the pixel zoom factors specified by PixelZoom, and may each be either positive or negative. A fragment representing group n,m is produced for each framebuffer pixel with one or more sample points that lie inside, or on the bottom or left boundary, of this rectangle. Each fragment so produced takes its associated data from the group and from the current raster position, in a manner consistent with the discussion in the Conversion to Fragments subsection of section 3.6.4 of the GL specification. All depth sample values are assigned the same value, taken either from the group (if it is a depth component group) or from the current raster position (if it is not).

A single pixel rectangle will generate multiple, perhaps very many fragments for the same framebuffer pixel, depending on the pixel zoom factors.

Bitmap Multisample Rasterization
[Insert at the end section 3.7]

If MULTISAMPLE_ARB is enabled, and SAMPLE_BUFFERS_ARB is a value of one, then bitmaps are rasterized using the following algorithm. If the current raster position is invalid, the bitmap is ignored. Otherwise, a screen-aligned array of pixel-size rectangles is constructed, with its lower-left corner at (Xrp,Yrp), and its upper right corner at (Xrp+w,Yrp+h), where w and h are the width and height of the bitmap. Rectangles in this array are eliminated if the corresponding bit in the bitmap is zero, and are retained otherwise. Bitmap rasterization produces a fragment for each framebuffer pixel with one or more sample points either inside or on the bottom or left edge of a retained rectangle.

Coverage bits that correspond to sample points either inside or on the bottom or left edge of a retained rectangle are 1, other coverage bits are 0. The associated data for each fragment are those associated with the current raster position. Once the fragments have been produced, the current raster position is updated exactly as it is in the single-sample rasterization case.

Additions to Chapter 4 of the 1.2.1 Specification (Per-Fragment Operations and the Frame Buffer)

Multisample Fragment Operations
[Insert after section 4.1.2]

This step modifies fragment alpha and coverage values based on the values of SAMPLE_ALPHA_TO_COVERAGE_ARB, SAMPLE_ALPHA_TO_ONE_ARB, SAMPLE_COVERAGE_ARB, SAMPLE_COVERAGE_VALUE_ARB, and SAMPLE_COVERAGE_INVERT_ARB. No changes to the fragment alpha or coverage values are made at this step if MULTISAMPLE_ARB is disabled, or if SAMPLE_BUFFERS_ARB is not a value of one.

SAMPLE_ALPHA_TO_COVERAGE_ARB, SAMPLE_ALPHA_TO_ONE_ARB, and SAMPLE_COVERAGE_ARB are enabled and disabled by calling Enable and Disable with <cap> specified as one of the three token values. All three values are queried by calling IsEnabled, with <cap> set to the desired token value. If SAMPLE_ALPHA_TO_COVERAGE_ARB is enabled, the fragment alpha value is used to generate a temporary coverage value, which is then ANDed with the fragment coverage value. Otherwise the fragment coverage value is unchanged at this point.

This specification does not require a specific algorithm for converting an alpha value to a temporary coverage value. It is intended that the number of 1's in the temporary coverage be proportional to the alpha value, with all 1's corresponding to the maximum alpha value, and all 0's corresponding to an alpha value of 0. It is also intended that the algorithm be pseudo-random in nature, to avoid image artifacts due to regular coverage sample locations. The algorithm can and probably should be different at different pixel locations. If it does differ, it should be defined relative to window, not screen, coordinates, so that rendering results are invariant with respect to window position.

Next, if SAMPLE_ALPHA_TO_ONE_ARB is enabled, fragment alpha is replaced by the maximum representable alpha value. Otherwise, fragment alpha value is not changed.

Finally, if SAMPLE_COVERAGE_ARB is enabled, the fragment coverage is ANDed with another temporary coverage. This temporary coverage is generated in the same manner as the one described above, but as a function of the value of SAMPLE_COVERAGE_VALUE_ARB. The function need not be identical, but it must have the same properties of proportionality and invariance. If SAMPLE_COVERAGE_INVERT_ARB is TRUE, the temporary coverage is inverted (all bit values are inverted) before it is ANDed with the fragment coverage.

The values of SAMPLE_COVERAGE_VALUE_ARB and SAMPLE_COVERAGE_INVERT_ARB are specified simultaneously by calling SampleCoverageARB, with <value> set to the desired coverage value, and <invert> set to TRUE or FALSE. <value> is clamped to [0,1] before being stored as SAMPLE_COVERAGE_VALUE_ARB. SAMPLE_COVERAGE_VALUE_ARB is queried by calling GetFloatv with <pname> set to SAMPLE_COVERAGE_VALUE_ARB. SAMPLE_COVERAGE_INVERT_ARB is queried by calling GetBooleanv with <pname> set to SAMPLE_COVERAGE_INVERT_ARB.

Multisample Fragment Operations
[Insert after section 4.1.8]

If the DrawBuffers mode is NONE, no change is made to any multisample or color buffer. Otherwise, fragment processing is as described below.

If MULTISAMPLE_ARB is enabled, and SAMPLE_BUFFERS_ARB is one, the stencil test, depth test, blending, and dithering operations are performed for each pixel sample, rather than just once for each fragment. Failure of the stencil or depth test results in termination of the processing of that sample, rather than discarding of the fragment. All operations are performed on the color, depth, and stencil values stored in the multisample buffer (to be described in a following section). The contents of the color buffers are not modified at this point.

Stencil, depth, blending, and dithering operations are performed for a pixel sample only if that sample's fragment coverage bit is a value of 1. If the corresponding coverage bit is 0, no operations are performed for that sample. Depth operations use the fragment depth value that is specific for each sample. The single fragment color value is used for all sample operations, however, as is the current stencil value.

If MULTISAMPLE_ARB is disabled, and SAMPLE_BUFFERS_ARB is one, the fragment may be treated exactly as described above, with optimization possible because the fragment coverage must be set to full coverage. Further optimization is allowed, however. An implementation may choose to identify a centermost sample, and to perform stencil and depth tests on only that sample. Regardless of the outcome of the stencil test, all multisample buffer stencil sample values are set to the appropriate new stencil value. If the depth test passes, all multisample buffer depth sample values are set to the depth of the fragment's centermost sample's depth value, and all multisample buffer color sample values are set to the color value of the incoming fragment. Otherwise, no change is made to any multisample buffer color or depth value.

After all operations have been completed on the multisample buffer, the color sample values are combined to produce a single color value, and that value is written into each color buffer that is currently enabled, based on the DrawBuffers mode. An implementation may defer the writing of the color buffer until a later time, but the state of the framebuffer must behave as if the color buffer was updated as each fragment was processed. The method of combination is not specified, though a simple average computed independently for each color component is recommended.

Fine Control of Multisample Buffer Updates
[Insert at the end of section 4.2.2]

When SAMPLE_BUFFERS_ARB is one, ColorMask, DepthMask, and StencilMask control the modification of values in the multisample buffer. The color mask has no effect on modifications to the color buffers. If the color mask is entirely disabled, the color sample values must still be combined (as described above) and the result used to replace the color values of the buffers enabled by DrawBuffers.

Clearing the Multisample Buffer
[Insert as a subsection for section 4.2.3]

The color samples of the multisample buffer are cleared when one or more color buffers are cleared, as specified by the Clear mask bit COLOR_BUFFER_BIT and the DrawBuffers mode. If the DrawBuffers mode is NONE, the color samples of the multisample buffer cannot be cleared.

Clear mask bits DEPTH_BUFFER_BIT and STENCIL_BUFFER_BIT indicate that the depth and stencil samples of the multisample buffer are to be cleared. If Clear mask bit DEPTH_BUFFER_BIT is specified, and if the DrawBuffers mode is not NONE, then the multisample depth buffer samples are cleared. Likewise, if Clear mask bit STENCIL_BUFFER_BIT is specified, and if the DrawBuffers mode is not NONE, then the multisample stencil buffer is cleared.

Reading Pixels
[These changes are made to the text in section 4.3.2, following the subheading Obtaining Pixels from the Framebuffer.]

Follow the sentence "If there is no depth buffer, the error INVALID_OPERATION occurs." with: If there is a multisample buffer (SAMPLE_BUFFERS_ARB is 1) then values are obtained from the depth samples in this buffer. It is recommended that the depth value of the centermost sample be used, though implementations may choose any function of the depth sample values at each pixel.

Follow the sentence "if there is no stencil buffer, the error INVALID_OPERATION occurs." with: If there is a multisample buffer, then values are obtained from the stencil samples in this buffer. It is recommended that the stencil value of the centermost sample be used, though implementations may choose any function of the stencil sample values at each pixel.

[This extension makes no change to the way that color values are obtained from the framebuffer.]

Additions to Chapter 5 of the 1.2.1 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

An additional group of state variables, MULTISAMPLE_BIT_ARB, is defined by this extension. When PushAttrib is called with bit MULTISAMPLE_BIT_ARB set, the multisample group of state variables is pushed onto the attribute stack. When PopAttrib is called, these state variables are restored to their previous values if they were pushed. Some multisample state is included in the ENABLE_BIT group as well. In order to avoid incompatibility with GL implementations that do not support SGIS_multisample, ALL_ATTRIB_BITS does not include MULTISAMPLE_BIT_ARB.

Additions to the GLX Specification

The parameter GLX_SAMPLE_BUFFERS_ARB is added to glXGetConfig. When queried, by calling glXGetConfig with <attrib> set to GLX_SAMPLE_BUFFERS_ARB, it returns the number of multisample buffers included in the visual. For a normal visual, the return value is zero. A return value of one indicates that a single multisample buffer is available. The number of samples per pixel is queried by calling glXGetConfig with <attrib> set to GLX_SAMPLES_ARB. It is understood that the number of color, depth, and stencil bits per sample in the multisample buffer are as specified by the GLX_*_SIZE parameters. It is also understood that there are no single-sample depth or stencil buffers associated with this visual -- the only depth and stencil buffers are those in the multisample buffer. GLX_SAMPLES_ARB is zero if GLX_SAMPLE_BUFFERS_ARB is zero.

glXChooseVisual accepts GLX_SAMPLE_BUFFERS_ARB in <attribList>, followed by the minimum number of multisample buffers that can be accepted. Visuals with the smallest number of multisample buffers that meets or exceeds the specified minimum number are preferred. Currently operation with more than one multisample buffer is undefined, so the returned value will be either zero or one.

glXChooseVisual accepts GLX_SAMPLES_ARB in <attribList>, followed by the minimum number of samples that can be accepted in the multisample buffer. Visuals with the smallest number of samples that meets or exceeds the specified minimum number are preferred.

If the color samples in the multisample buffer store fewer bits than are stored in the color buffers, this fact will not be reported accurately. Presumably a compression scheme is being employed, and is expected to maintain an aggregate resolution equal to that of the color buffers.

GLX Protocol

TBD

Additions to the WGL Specification

The parameter WGL_SAMPLE_BUFFERS_ARB is added to wglGetPixelFormatAttrib*v. When queried, by calling wglGetPixelFormatAttrib*v with <piAttributes> set to WGL_SAMPLE_BUFFERS_ARB, it returns the number of multisample buffers included in the pixel format. For a normal pixel format, the return value is zero. A return value of one indicates that a single multisample buffer is available. The number of samples per pixel is queried by calling wglGetPixelFormatAttrib*v with <piAttributes> set to WGL_SAMPLES_ARB. It is understood that the number of color, depth, and stencil bits per sample in the multisample buffer are as specified by the WGL_*_SIZE parameters. It is also understood that there are no single-sample depth or stencil buffers associated with this visual -- the only depth and stencil buffers are those in the multisample buffer. WGL_SAMPLES_ARB is zero if WGL_SAMPLE_BUFFERS_ARB is zero.

wglChoosePixelFormatEXT accepts WGL_SAMPLE_BUFFERS_ARB in <piAttribIList> and <pfAttribIList> with the corresponding value set to the minimum number of multisample buffers that can be accepted. Pixel formats with the smallest number of multisample buffers that meets or exceeds the specified minimum number are preferred. Currently operation with more than one multisample buffer is undefined, so the returned value will be either zero or one.

If the color samples in the multisample buffer store fewer bits than are stored in the color buffers, this fact will not be reported accurately. Presumably a compression scheme is being employed, and is expected to maintain an aggregate resolution equal to that of the color buffers.

Errors

    INVALID_OPERATION is generated if SampleCoverageARB is called
    between the execution of Begin and the execution of the
    corresponding End.

New State

Get Value	Get Command	Type	Initial Value	Attribute
MULTISAMPLE_ARB	IsEnabled	B	TRUE	multisample/enable
SAMPLE_ALPHA_TO_COVERAGE_ARB	IsEnabled	B	FALSE	multisample/enable
SAMPLE_ALPHA_TO_ONE_ARB	IsEnabled	B	FALSE	multisample/enable
SAMPLE_COVERAGE_ARB	IsEnabled	B	FALSE	multisample/enable
SAMPLE_COVERAGE_VALUE_ARB	GetFloatv	R+	1	multisample
SAMPLE_COVERAGE_INVERT_ARB	GetBooleanv	B	FALSE	multisample

New Implementation Dependent State

Get Value	Get Command	Type	Minimum Value
SAMPLE_BUFFERS_ARB	GetIntegerv	Z+	0
SAMPLES_ARB	GetIntegerv	Z+	0

Conformance Testing

    TBD

Revision History

```
    09/20/1999  0.1
        - First ARB draft based on the original SGI draft.

    10/1/1999   0.2
        - Added query for the number of passes.

    11/8/1999   0.3
        - Fixed numerous typos reported by E&S.

    12/7/1999   0.4
        - Removed the multiple pass feature.
        - Resolved the working group issues at the ARB meeting.
        - Added language that stated that SAMPLE_BUFFERS_ARB is the
          same value as either GLX_SAMPLE_BUFFERS_ARB or
          WGL_SAMPLE_BUFFERS_ARB.

    12/15/1999  0.5
        - Added back in the statement about ALL_ATTRIB_BITS not
          including MULTISAMPLE_BIT_ARB.
```

Name Strings

    ARB_multitexture

Name Strings

    GL_ARB_multitexture

Status

    Complete. Approved by ARB on 9/15/1998

NOTE: This extension no longer has its own specification document, since it has been included in the OpenGL 1.2.1 Specification (downloadable from www.opengl.org). Please refer to the 1.2.1 Specification for more information.

Name

    ARB_occlusion_query

Name Strings

    GL_ARB_occlusion_query

Notice

IP Status

HP has claimed that they hold IP around use of this extension. HP has committed to releasing rights to this IP to the ARB if the functionality is included in OpenGL (April 10, 2003).

Status

Approved by the ARB (version 1.0), June 10, 2003, pending further minor revisions

Version

```
NVIDIA Date: June 24, 2003
$Id: //sw/main/docs/OpenGL/specs/GL_ARB_occlusion_query.txt#2 $
```

Number

ARB Extension #29

Dependencies

Written based on the wording of the OpenGL 1.4 specification.

HP_occlusion_test affects the definition of this extension.

Overview

This extension defines a mechanism whereby an application can query the number of pixels (or, more precisely, samples) drawn by a primitive or group of primitives.

The primary purpose of such a query (hereafter referred to as an "occlusion query") is to determine the visibility of an object. Typically, the application will render the major occluders in the scene, then perform an occlusion query for the bounding box of each detail object in the scene. Only if said bounding box is visible, i.e., if at least one sample is drawn, should the corresponding object be drawn.

```
    The earlier HP_occlusion_test extension defined a similar mechanism,
    but it had two major shortcomings.

    - It returned the result as a simple GL_TRUE/GL_FALSE result, when in
      fact it is often useful to know exactly how many samples were
      drawn.

    - It provided only a simple "stop-and-wait" model for using multiple
      queries.  The application begins an occlusion test and ends it;
      then, at some later point, it asks for the result, at which point
      the driver must stop and wait until the result from the previous
      test is back before the application can even begin the next one.
      This is a very simple model, but its performance is mediocre when
      an application wishes to perform many queries, and it eliminates
      most of the opportunities for parallelism between the CPU and GPU.

    This extension solves both of those problems.  It returns as its
    result the number of samples that pass the depth and stencil tests,
    and it encapsulates occlusion queries in "query objects" that allow
    applications to issue many queries before asking for the result of
    any one.  As a result, they can overlap the time it takes for the
    occlusion query results to be returned with other, more useful work,
    such as rendering other parts of the scene or performing other
    computations on the CPU.

    There are many situations where a pixel/sample count, rather than a
    boolean result, is useful.

    - Objects that are visible but cover only a very small number of
      pixels can be skipped at a minimal reduction of image quality.

    - Knowing exactly how many pixels an object might cover may help the
      application decide which level-of-detail model should be used.  If
      only a few pixels are visible, a low-detail model may be
      acceptable.

    - "Depth peeling" techniques, such as order-independent transparency,
      need to know when to stop rendering more layers; it is difficult to
      determine a priori how many layers are needed.  A boolean result
      allows applications to stop when more layers will not affect the
      image at all, but this will likely result in unacceptable
      performance.  Instead, it makes more sense to stop rendering when
      the number of pixels in each layer falls below a given threshold.

    - Occlusion queries can replace glReadPixels of the depth buffer to
      determine whether (for example) a light source is visible for the
      purposes of a lens flare effect or a halo to simulate glare.  Pixel
      counts allow you to compute the percentage of the light source that
      is visible, and the brightness of these effects can be modulated
      accordingly.
```

Issues

How is this extension different from NV_occlusion_query?

The following changes have been made.

- A "target" parameter has been added. Only one target exists at present, SAMPLES_PASSED_ARB, but future extensions could add additional types of queries.
- Terminology changed slightly. "Pixel" was being used incorrectly, where "fragment" or "sample" would be more accurate.
- Various NVIDIA-specific references have been removed.
- Interactions with multisample have been changed slightly to allow implementations based on either a sample count or a fragment count. The result is returned in units of samples.
- Clarified that using an id of zero is illegal.
- Added missing spec language for IsQuery entry point.
- General language, issues, etc. cleanup.
- HP_occlusion_test is no longer required.
- Modified the minimum required counter bits to be dependent on the implementation's maximum viewport or the value 0
- Clarified that active query state is per target server state. This implies that a loop of QUERY_RESULT_AVAILABLE_ARB will return TRUE in finite time. NV_occlusion_query asked that the application flush to prevent an infinite loop.
- Clarified the behavior of the async QUERY_RESULT_AVAILABLE_ARB command.
- When the count of samples that pass the occlusion query overflows, the value should saturate.

Should we use an object-based interface?

RESOLVED: Yes, this makes the interface much simpler, and it is friendly for indirect rendering.

What is the difference between a "query object" and an "occlusion query"?

"Occlusion query" is a synonym for "query object used with target SAMPLES_PASSED".

Should we offer a way to poll whether an occlusion query has completed and its result is available?

RESOLVED. Yes, this allows applications to use CPU time that might have been spent spinning more usefully. However, the polling method introduced in the NV_occlusion_query spec allowed for a potential infinite loop if the application does not do a flush. This version of the spec clarifies the behavior which now makes such a flush unnecessary.

Is GetQueryObjectuivARB necessary?

RESOLVED: Yes, it makes using a 32-bit count less painful.

Should there be a limit on how many queries can be outstanding?

RESOLVED: No. This would make the extension much more difficult to spec and use. Allowing this does not add any significant implementation burden; and even if drivers have some internal limit on the number of outstanding queries, it is not expected that applications will need to know this to achieve optimal or near-optimal performance.

What happens if BeginQueryARB is called when a query is already outstanding for a different object on the same target?

RESOLVED: This is an INVALID_OPERATION error.

What happens if BeginQueryARB is called with an ID of a query that is already in progress?

RESOLVED: This is also an INVALID_OPERATION error.

What parameters should EndQueryARB have?

RESOLVED: Just a target. It doesn't need to take an "id" argument, since this would be redundant -- only one query can be active for any given target at a given time.

How many bits should we require the samples-passed count to be, at minimum?

RESOLVED. The largest minimum that can be required of a GL implementation is 32, the minimum bit width of an int or uint.

The minimum number of bits required for the samples-passed count will be dependent on the implementation's maximum viewport size. In order to allow for two overdraws in the case of only one sample buffer, the minimum counter precision (n) will be determined by:

```
n = min (32 , ceil (log2 (maxViewportWidth x maxViewportHeight x
                1 sample x 2 overdraws) ) )
```

An implementation can either set QUERY_COUNTER_BITS_ARB to the value 0, or to some number greater than or equal to n. If an implementation returns 0 for QUERY_COUNTER_BITS_ARB, then the occlusion queries will always return that zero samples passed the occlusion test, and so an application should not use occlusion queries on that implementation.

Note that other targets may come along in the future that require more or fewer bits.

What should we do about overflows?

RESOLVED: Overflows are required to saturate, though it is expected that several current implementations will not conform to this requirement.

The ideal behavior is to saturate. This ensures that you always get a "large" result when you render many samples. It also ensures that apps which want a boolean test can do this without worrying about the rare case where the result ends up exactly at zero after wrapping.

Either way, it's unlikely that this matters much as long as a sufficient number of bits are required.

What is the interaction with multisample?

RESOLVED: We count samples, not pixels -- even if MULTISAMPLE is disabled but SAMPLE_BUFFERS is 1.

A given fragment may have anywhere between zero and SAMPLES of its samples covered. Ideally, the samples-passed count would be incremented by the precise number of samples, but we permit implementations to instead increment the samples-passed count by SAMPLES if at least one sample in a given fragment is covered.

Note that the depth/stencil test optimization whereby implementations may choose to depth test at only one of the samples when MULTISAMPLE is disabled does not cause this to become ill-specified, because we are counting the number of samples that are still alive _after_ the depth test stage. The particular mechanism used to decide whether to kill or keep those samples is not relevant.

Exactly what stage in the pipeline are we counting samples at?

RESOLVED: We are counting immediately after _both_ the depth and stencil tests, i.e., samples that pass both. Note that the depth test comes after the stencil test, so to say that it is the number that pass the depth test is sufficient; though it is often conceptually helpful to think of the depth and stencil tests as being combined, because the depth test's result impacts the stencil operation used.

Is it guaranteed that occlusion queries return in order?

RESOLVED: Yes. It makes sense to do this. If occlusion test X occurred before occlusion query Y, and the driver informs the app that occlusion query Y is done, the app can infer that occlusion query X is also done. For applications that do poll, this allows them to do so with less effort.

Will polling a query without a Flush possibly cause an infinite loop?

RESOLVED: No. An infinite loop was possible in the original NV_occlusion_query spec if an application did not perform a flush prior to polling. This behavior was removed in this version of the spec as it violated language in the core GL spec.

Instead of allowing for an infinite loop, performing a QUERY_RESULT_AVAILABLE_ARB will perform a flush if the result is not ready yet on the first time it is queried. This ensures that the async query will return true in finite time.

This behavior is not a significant performance loss over the original version of the spec. A flush would need to be performed at some point anyway and the flush performed when QUERY_RESULT_AVAILABLE_ARB is requested will only occur *if the result is not back yet*.

What should be the interaction between this spec and HP_occlusion_test?

RESOLVED: Whereas NV_occlusion_query required that you implement HP_occlusion_test, and even went so far as to specify the precise behavior of HP_occlusion_test (since the HP_occlusion_test spec did not contain those details), this spec does not. This spec explains the interaction with HP_occlusion_test, but does not attempt to double as a spec for that extension.

What happens if HP_occlusion_test and ARB_occlusion_query usage is overlapped?

RESOLVED: The two can be overlapped safely. Counting is enabled if either an occlusion query is active *or* OCCLUSION_TEST_HP is enabled. The alternative (producing an error) does not work -- it would require that PopAttrib be capable of producing an error, which would be rather problematic.

Note that BeginQueryARB, not EndQueryARB, resets the sample count (and therefore the occlusion test result). This can avoid certain types of strange behavior where an occlusion query's samples-passed count does not always correspond to the samples rendered during the occlusion query. The spec would make sense the other way, but the behavior would be strange.

Should there be a "target" parameter to BeginQueryARB?

RESOLVED: Yes. Whereas NV_occlusion_query wasn't trying to solve a problem beyond simple occlusion queries, this extension creates a framework useful for future queries.

Does GenQueriesARB need a "target" parameter?

RESOLVED: No. A query can be reused any number of times with any targets.

How can one ask for the currently active query?

RESOLVED: A new entry point has been added to query information about a given query target. This makes it unnecessary to add two new enumerants (# of bits and current query ID) for each new target that is introduced.

Are query objects shareable between multiple contexts?

RESOLVED: No. Query objects are lightweight and we normally share large data across contexts. Also, being able to share query objects across contexts is not particularly useful. In order to do the async query across contexts, a query on one context would have to be finished before the other context could query it.

What happens when an app begins a query on a target, ends it, begins a query on the same target with the same id, ends it, and then tries to retrieve data about the query using GetQueryObjecti[u]vARB? Which query does the GetQueryObjecti[u]vARB return results for?

RESOLVED. In this case, the result retrieved from GetQueryObjecti[u]vARB will be from the last query on that target and id. The result returned from GetQueryObjecti[u]vARB will always be from the last BeginQueryARB/EndQueryARB pair on that target and id.

Is this extension useful for saving geometry, fill rate, or both?

The answer to this question is to some extent implementation-dependent, but it is expected that it is most useful for reducing geometry workload, and less so for fill rate.

For the cost of rendering a bounding box, you can potentially save rendering a normal object. A bounding box consists of only 12 triangles, whereas the original object might have contained thousands or even millions of triangles.

Using bounding box occlusion queries may either help or hurt in fill-limited situations, because rendering the pixels of a bounding box is not free. In most situations, a bounding box will probably have more pixels than the original object. Those pixels can probably be rendered more quickly, though, since they involve only Z reads (no Z writes or color traffic), and they need not be textured or otherwise shaded.

In multipass rendering situations, however, occlusion queries can almost always save fill rate, because wrapping an object with an occlusion query is generally cheap. See "Usage Examples" for an illustration.

What can be said about guaranteeing correctness when using occlusion queries, especially as it relates to invariance?

Invariance is critical to guarantee the correctness of occlusion queries. If occlusion queries go through a different code path than standard rendering, the fragments rendered may be different.

However, the invariance issues are difficult at best to solve. Because of the vagaries of floating-point precision, it is difficult to guarantee that rendering a bounding box will render at least as many pixels with equal or smaller Z values than the object itself would have rendered.

Likewise, many other aspects of rendering state tend to be different when performing an occlusion query. Color and depth writes are typically disabled, as are texturing, vertex programs, and any fancy per-fragment math. So unless all these features have guarantees of invariance themselves (unlikely at best), requiring invariance for ARB_occlusion_query would be futile.

In general, implementations are recommended to be fully invariant with respect to whether any given type of query is active, insofar as it is possible. That is, having an occlusion query active should not affect the operation of any other stage of the pipeline. Following this rule is essential to numerous occlusion query algorithms working correctly. However, to permit implementations where this feature is implemented in software, this rule is only a recommendation, not a requirement.

Another unrelated problem that can threaten correctness is near and far clipping. The bounding box of an object may penetrate the near clip plane, even though the original object may not have. In such a circumstance, a bounding box occlusion query may produce an incorrect result. Whenever you design an algorithm using occlusion queries, it is best to be careful about the near and far clip planes.

How can frame-to-frame coherency help applications using this extension get even higher performance?

Usually, if an object is visible one frame, it will be visible the next frame, and if it is not visible, it will not be visible the next frame.

Of course, for most applications, "usually" isn't good enough. It is undesirable, but acceptable, to render an object that *isn't* visible, because that only costs performance. It is generally unacceptable to *not* render an object that *is* visible.

The simplest approach is that visible objects should be checked every N frames (where, say, N=5) to see if they have become occluded, while objects that were occluded last frame must be rechecked again in the current frame to guarantee that they are still occluded. This will reduce the number of wasteful occlusion queries by almost a factor of N.

Other, more complicated techniques exist but are beyond the scope of this extension document.

Do occlusion queries make other visibility algorithms obsolete?

No.

Occlusion queries are helpful, but they are not a cure-all. They should be only one of many items in your bag of tricks to decide whether objects are visible or invisible. They are not an excuse to skip frustum culling, or precomputing visibility using portals for static environments, or other standard visibility techniques.

New Procedures and Functions

```
    void GenQueriesARB(sizei n, uint *ids);
    void DeleteQueriesARB(sizei n, const uint *ids);
    boolean IsQueryARB(uint id);
    void BeginQueryARB(enum target, uint id);
    void EndQueryARB(enum target);
    void GetQueryivARB(enum target, enum pname, int *params);
    void GetQueryObjectivARB(uint id, enum pname, int *params);
    void GetQueryObjectuivARB(uint id, enum pname, uint *params);
```

New Tokens

Accepted by the <target> parameter of BeginQueryARB, EndQueryARB, and GetQueryivARB:

```
        SAMPLES_PASSED_ARB                            0x8914
```

Accepted by the <pname> parameter of GetQueryivARB:

```
        QUERY_COUNTER_BITS_ARB                        0x8864
        CURRENT_QUERY_ARB                             0x8865
```

Accepted by the <pname> parameter of GetQueryObjectivARB and GetQueryObjectuivARB:

```
        QUERY_RESULT_ARB                              0x8866
        QUERY_RESULT_AVAILABLE_ARB                    0x8867
```

Additions to Chapter 2 of the OpenGL 1.4 Specification (OpenGL Operation)

Modify Section 2.1, OpenGL Fundamentals (p. 4)

(modify fourth paragraph, p. 4) It also means that queries and pixel read operations return state consistent with complete execution of all previously invoked GL commands, except where explicitly specified otherwise

Additions to Chapter 3 of the OpenGL 1.4 Specification (Rasterization)

None.

Additions to Chapter 4 of the OpenGL 1.4 Specification (Per-Fragment Operations and the Frame Buffer)

Add a new section "Occlusion Queries" between sections 4.1.6 and 4.1.7:

"4.1.6A Occlusion Queries

Occlusion queries can be used to track the number of fragments or samples that pass the depth test.

Occlusion queries are associated with query objects. The command

```
  void GenQueriesARB(sizei n, uint *ids);
```

returns <n> previously unused query object names in <ids>. These names are marked as used, but no object is associated with them until the first time they are used by BeginQueryARB. Query objects contain one piece of state, an integer result value. This result value is initialized to zero when the object is created. Any positive integer except for zero (which is reserved for the GL) is a valid query object name.

Query objects are deleted by calling

```
  void DeleteQueriesARB(sizei n, const uint *ids);
```

<ids> contains <n> names of query objects to be deleted. After a query object is deleted, its name is again unused. Unused names in <ids> are silently ignored.

An occlusion query can be started and finished by calling

```
  void BeginQueryARB(enum target, uint id);
  void EndQueryARB(enum target);
```

where <target> is SAMPLES_PASSED_ARB. If BeginQueryARB is called with an unused <id>, that name is marked as used and associated with a new query object. If BeginQueryARB is called while another query is already in progress with the same target, an INVALID_OPERATION error is generated. If EndQueryARB is called while no query with the same target is in progress, an INVALID_OPERATION error is generated. Calling either GenQueriesARB or DeleteQueriesARB while any query of any target is active causes an INVALID_OPERATION error to be generated.

BeginQueryARB with a <target> of SAMPLES_PASSED_ARB resets the current samples-passed count to zero and sets the query active state to TRUE and the active query id to <id>. EndQueryARB with a target of SAMPLES_PASSED_ARB initializes a copy of the current samples-passed count into the active occlusion query object's results value, sets the active occlusion query object's result available to FALSE, sets the query active state to FALSE, and the active query id to 0.

If BeginQueryARB is called with an <id> of zero, or where <id> is the name of a query currently in progress, an INVALID_OPERATION error is generated.

When an occlusion query is active, the samples-passed count increases by a certain quantity for each fragment that passes the depth test. If the value of SAMPLE_BUFFERS is 0, then the samples-passed count increases by 1 for each fragment. If the value of SAMPLE_BUFFERS is 1, then the samples-passed count increases by the number of samples whose coverage bit is set. However, implementations, at their discretion, are allowed to instead increase the samples-passed count by the value of SAMPLES if any sample in the fragment is covered.

If the samples-passed count overflows, i.e., exceeds the value 2^n-1 (where n is the number of bits in the samples-passed count), its value becomes undefined. It is recommended, but not required, that implementations handle this overflow case by saturating at 2^n-1 and incrementing no further.

The necessary state is a single bit indicating whether an occlusion query is active, the identifier of the currently active occlusion query, and a counter keeping track of the number of samples that have passed."

Additions to Chapter 5 of the OpenGL 1.4 Specification (Special Functions)

Add to the end of Section 5.4 "Display Lists":

"DeleteQueriesARB, GenQueriesARB, IsQueryARB, GetQueryivARB, GetQueryObjectivARB, and GetQueryObjectuivARB are not compiled into display lists but are executed immediately."

Additions to Chapter 6 of the OpenGL 1.4 Specification (State and State Requests)

Add a new section 6.1.13 "Occlusion Queries":

"The command

```
  boolean IsQueryARB(uint id);
```

returns TRUE if <id> is the name of a query object. If <id> is zero, or if <id> is a non-zero value that is not the name of a query object, IsQueryARB returns FALSE.

Information about a query target can be queried with the command

```
  void GetQueryivARB(enum target, enum pname, int *params);
```

If <pname> is CURRENT_QUERY_ARB, the name of the currently active query for <target>, or zero if no query is active, will be placed in <params>.

If <pname> is QUERY_COUNTER_BITS_ARB, the number of bits in the counter for <target> will be placed in <params>. The minimum number of query counter bits allowed is a function of the implementation's maximum viewport dimensions (MAX_VIEWPORT_DIMS). If the counter is non-zero, then the counter must be able to represent at least two overdraws for every pixel in the viewport using only one sample buffer. The formula to compute the allowable minimum value is below (where n is the minimum number of bits):

```
  n = (min (32, ceil (log2 (maxViewportWidth x maxViewportHeight x 2) ) ) )
       or 0
```

If the value of n is 0, then the result from GetQueryiv(SAMPLES_PASSED_ARB) will always return 0,

```
    The state of a query object can be queried with the commands

      void GetQueryObjectivARB(uint id, enum pname, int *params);
      void GetQueryObjectuivARB(uint id, enum pname, uint *params);

    If <id> is not the name of a query object, or if the query object
    named by <id> is currently active, then an INVALID_OPERATION error is
    generated.

    If <pname> is QUERY_RESULT_ARB, then the query object's result value
    is placed in <params>.

    Often, query object results will be returned asynchronously with
    respect to the host processor's operation.  As a result, sometimes,
    if a result is queried, the host must wait until the result is back.
    If <pname> is QUERY_RESULT_AVAILABLE_ARB, the value placed in <params>
    indicates whether or not such a wait would occur if the result of
    that query object were to be queried presently.  A result of TRUE
    means no wait would be required; a result of FALSE means that some
    wait would occur.  It must always be true that if the result for
    one query is available, the result for all previous queries must
    also be available at that point in time.

    Querying the state for a given occlusion query forces that occlusion
    query to complete within a finite amount of time.

    If multiple queries are issued on the same target and id prior to
    calling GetQueryObject[u]iVARB, the result returned will always be
    from the last query issued.  The results from any queries before
    the last one will be lost if the results are not retrieved before
    starting a new query on the same target and id."
```

Dependencies on HP_occlusion_test

```
    When GetIntegerv is called with <pname> of OCCLUSION_TEST_RESULT_HP,
    the current samples-passed count is reset to zero.  The occlusion
    test result is TRUE when the samples-passed count is nonzero, and
    FALSE when it is zero.  Sample counting is active (i.e. the samples-
    passed count increases as fragments are drawn) whenever either an
    occlusion query is active *or* OCCLUSION_TEST_HP is enabled.
```

GLX Protocol

```
    Seven new GL commands are added.

    The following two rendering commands are sent to the server as part
    of a glXRender request:

        BeginQueryARB
            2           12              rendering command length
            2           ????            rendering command opcode
            4           ENUM            target
            4           CARD32          id
        EndQueryARB
            2           8               rendering command length
            2           ????            rendering command opcode
            4           ENUM            target

    The remaining fivecommands are non-rendering commands.  These
    commands are sent separately (i.e., not as part of a glXRender or
    glXRenderLarge request), using the glXVendorPrivateWithReply
    request:

        DeleteQueriesARB
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           4+n             request length
            4           ????            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           INT32           n
            n*4         LISTofCARD32    ids

        GenQueriesARB
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           4               request length
            4           ????            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           INT32           n
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           n               reply length
            24                          unused
            n*4         LISTofCARD322   queries

        IsQueryARB
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           4               request length
            4           ????            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           CARD32          id
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           0               reply length
            4           BOOL32          return value
            20                          unused
            1           1               reply
        GetQueryivARB
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           5               request length
            4           ????            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           ENUM            target
            4           ENUM            pname
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           m               reply length, m=(n==1?0:n)
            4                           unused
            4           CARD32          n

            if (n=1) this follows:

            4           INT32           params
            12                          unused

            otherwise this follows:

            16                          unused
            n*4         LISTofINT32     params

        GetQueryObjectivARB
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           5               request length
            4           ????            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           CARD32          id
            4           ENUM            pname
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           m               reply length, m=(n==1?0:n)
            4                           unused
            4           CARD32          n

            if (n=1) this follows:

            4           INT32           params
            12                          unused

            otherwise this follows:

            16                          unused
            n*4         LISTofINT32     params
        GetQueryObjectuivARB
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           5               request length
            4           ????            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           CARD32          id
            4           ENUM            pname
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           m               reply length, m=(n==1?0:n)
            4                           unused
            4           CARD32          n

            if (n=1) this follows:

            4           CARD32          params
            12                          unused

            otherwise this follows:

            16                          unused
            n*4         LISTofCARD32    params
```

Errors

```
    The error INVALID_VALUE is generated if GenQueriesARB is called where
    <n> is negative.

    The error INVALID_VALUE is generated if DeleteQueriesARB is called
    where <n> is negative.

    The error INVALID_OPERATION is generated if GenQueriesARB or
    DeleteQueriesARB is called when a query of any target is active.

    The error INVALID_ENUM is generated if BeginQueryARB, EndQueryARB, or
    GetQueryivARB is called where <target> is not SAMPLES_PASSED_ARB.

    The error INVALID_OPERATION is generated if BeginQueryARB is called
    when a query of the given <target> is already active.

    The error INVALID_OPERATION is generated if EndQueryARB is called
    when a query of the given <target> is not active.

    The error INVALID_OPERATION is generated if BeginQueryARB is called
    where <id> is zero.

    The error INVALID_OPERATION is generated if BeginQueryARB is called
    where <id> is is the name of a query currently in progress.

    The error INVALID_ENUM is generated if GetQueryivARB is called where
    <pname> is not QUERY_COUNTER_BITS_ARB or CURRENT_QUERY_ARB.
```

    The error INVALID_OPERATION is generated if GetQueryObjectivARB or
    GetQueryObjectuivARB is called where <id> is not the name of a query
    object.

    The error INVALID_OPERATION is generated if GetQueryObjectivARB or
    GetQueryObjectuivARB is called where <id> is the name of a currently
    active query object.

    The error INVALID_ENUM is generated if GetQueryObjectivARB or
    GetQueryObjectuivARB is called where <pname> is not QUERY_RESULT_ARB
    or QUERY_RESULT_AVAILABLE_ARB.

    The error INVALID_OPERATION is generated if any of the commands
    defined in this extension is executed between the execution of Begin
    and the corresponding execution of End.

New State

(table 6.18, p. 233)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
-	B	-	FALSE	query active	4.1.6A	-
CURRENT_QUERY_ARB	Z+	GetQueryiv	0	active query ID	4.1.6A	-
-	Z+	-	0	samples-passed count	4.1.6A	-

New Implementation Dependent State

(table 6.29, p. 224) Add the following entry:

Get Value	Type	Get Command	Minimum Value	Description	Sec	Attribute
QUERY_COUNTER_BITS_ARB	Z+	GetQueryiv	see 6.1.13	Number of bits in query counter	6.1.13	-

Revision History

    none yet

Usage Examples

Here is some rough sample code that illustrates how this extension can be used.

```
GLuint queries[N];
GLuint sampleCount;
GLint available;
GLuint bitsSupported;

// check to make sure functionality is supported
glGetQueryiv(GL_QUERY_COUNTER_BITS_ARB, &bitsSupported);
if (bitsSupported == 0) {
    // render scene without using occlusion queries
}

glGenQueriesARB(N, queries);
...
// before this point, render major occluders
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
glDepthMask(GL_FALSE);
// also disable texturing and any fancy shaders
for (i = 0; i < N; i++) {
    glBeginQueryARB(GL_SAMPLES_PASSED_ARB, queries[i]);
    // render bounding box for object i
    glEndQueryARB(GL_SAMPLES_PASSED_ARB);
}

glFlush();

// Do other work until "most" of the queries are back, to avoid
// wasting time spinning
i = N*3/4; // instead of N-1, to prevent the GPU from going idle
do {
    DoSomeStuff();
    glGetQueryObjectivARB(queries[i],
                          GL_QUERY_RESULT_AVAILABLE_ARB,
                          &available);
} while (!available);

glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
glDepthMask(GL_TRUE);
// reenable other state, such as texturing
for (i = 0; i < N; i++) {
    glGetQueryObjectuivARB(queries[i], GL_QUERY_RESULT_ARB,
                           &sampleCount);
    if (sampleCount > 0) {
        // render object i
    }
}
```

Here is some rough sample code for a simple multipass rendering application that does not use occlusion queries.

```
    for (i = 0; i < N; i++) {
        // First rendering pass
        glDisable(GL_BLEND);
        glDepthFunc(GL_LESS);
        glDepthMask(GL_TRUE);
        // configure shader 0
        // render object i

        // Second rendering pass
        glEnable(GL_BLEND);
        glBlendFunc(...);
        glDepthFunc(GL_EQUAL);
        glDepthMask(GL_FALSE);
        // configure shader 1
        // render object i
    }
```

Here is the previous example, enhanced using occlusion queries.

```
    GLuint queries[N];
    GLuint sampleCount;

    glGenQueriesARB(N, queries);
    ...
    // First rendering pass plus almost-free visibility checks
    glDisable(GL_BLEND);
    glDepthFunc(GL_LESS);
    glDepthMask(GL_TRUE);
    // configure shader 0
    for (i = 0; i < N; i++) {
        glBeginQueryARB(GL_SAMPLES_PASSED_ARB, queries[i]);
        // render object i
        glEndQueryARB(GL_SAMPLES_PASSED_ARB);
    }

    // Second pass only on objects that were visible
    glEnable(GL_BLEND);
    glBlendFunc(...);
    glDepthFunc(GL_EQUAL);
    glDepthMask(GL_FALSE);
    // configure shader 1
    for (i = 0; i < N; i++) {
        glGetQueryObjectuivARB(queries[i], GL_QUERY_RESULT_ARB,
                               &sampleCount);
        if (sampleCount > 0) {
            // render object i
        }
    }
```

Name

    ARB_pixel_buffer_object

Name Strings

    GL_ARB_pixel_buffer_object

Status

    Complete. Approved by ARB on December 7, 2004.

Contributors

    Ralf Biermann
    Nick Carter
    Derek Cornish
    Matt Craighead
    Mark Kilgard
    Dale Kirkland
    Jon Leech
    Brian Paul
    Thomas Roell
    Ian Romanick
    Jeremy Sandmel

Contact

    Mark J. Kilgard, NVIDIA Corporation (mjk 'at' nvidia.com)
    Ralf Biermann, NVIDIA Corporation (rbiermann 'at' nvidia.com)
    Derek Cornish, NVIDIA Corporation (dcornish 'at' nvidia.com)

IP Status

    None.

Version

    Last Modified Date: December 8, 2004
    Revision: 1.0

Number

    ARB Extension #42

Dependencies

    Written based on the wording of the OpenGL 2.0 specification.

    Assumes support for (at least) OpenGL 1.5 or the
    ARB_vertex_buffer_object extension.

    NV_pixel_data_range affects the definition of this extension.

    EXT_pixel_buffer_object interacts with this extension.

Overview

This extension expands on the interface provided by the ARB_vertex_buffer_object extension (and later integrated into OpenGL 1.5) in order to permit buffer objects to be used not only with vertex array data, but also with pixel data. The intent is to provide more acceleration opportunities for OpenGL pixel commands.

While a single buffer object can be bound for both vertex arrays and pixel commands, we use the designations vertex buffer object (VBO) and pixel buffer object (PBO) to indicate their particular usage in a given situation.

Recall that buffer objects conceptually are nothing more than arrays of bytes, just like any chunk of memory. ARB_vertex_buffer_object allows GL commands to source data from a buffer object by binding the buffer object to a given target and then overloading a certain set of GL commands' pointer arguments to refer to offsets inside the buffer, rather than pointers to user memory. An offset is encoded in a pointer by adding the offset to a null pointer.

This extension does not add any new functionality to buffer objects themselves. It simply adds two new targets to which buffer objects can be bound: GL_PIXEL_PACK_BUFFER and GL_PIXEL_UNPACK_BUFFER. When a buffer object is bound to the GL_PIXEL_PACK_BUFFER target, commands such as glReadPixels pack (write) their data into a buffer object. When a buffer object is bound to the GL_PIXEL_UNPACK_BUFFER target, commands such as glDrawPixels and glTexImage2D unpack (read) their data from a buffer object.

There are a several approaches to improve graphics performance with PBOs. Some of the most interesting approaches are:

- Streaming texture updates: If the application uses glMapBuffer/glUnmapBuffer to write its data for glTexSubImage into a buffer object, at least one of the data copies usually required to download a texture can be eliminated, significantly increasing texture download performance.

- Streaming draw pixels: When glDrawPixels sources client memory, OpenGL says the client memory can be modified immediately after the glDrawPixels command returns without disturbing the drawn image. This typically necessitates unpacking and copying the image prior to glDrawPixels returning. However, when using glDrawPixels with a pixel pack buffer object, glDrawPixels may return prior to image unpacking because future modification of the buffer data requires explicit commands (glMapBuffer, glBufferData, or glBufferSubData).

- Asynchronous glReadPixels: If an application needs to read back a number of images and process them with the CPU, the existing GL interface makes it nearly impossible to pipeline this operation. The driver will typically send the hardware a readback command when glReadPixels is called, and then wait for all of the data to be available before returning control to the application. Then, the application can either process the data immediately or call glReadPixels again; in neither case will the readback overlap with the processing. If the application issues several readbacks into several buffer objects, however, and then maps each one to process its data, then the readbacks can proceed in parallel with the data processing.

- Render to vertex array: The application can use a fragment program to render some image into one of its buffers, then read this image out into a buffer object via glReadPixels. Then, it can use this buffer object as a source of vertex data.

Issues

1) *How does this extension relate to ARB_vertex_buffer_object?*

It builds on the ARB_vertex_buffer_object framework by adding two new targets that buffers can be bound to.

2) *How does this extension relate to NV_pixel_data_range?*

This extension relates to NV_pixel_data_range in the same way that ARB_vertex_buffer_object relates to NV_vertex_array_range. To paraphrase the ARB_vertex_buffer_object spec, here are the main differences:

- Applications are no longer responsible for memory management and synchronization.

- Applications may still access high-performance memory directly, but this is optional, and such access is more restricted.

- Buffer changes (glBindBuffer) are generally expected to be very lightweight, rather than extremely heavyweight (glPixelDataRangeNV).

- A platform-specific allocator such as wgl/glXAllocateMemoryNV is no longer required.

3) *Can a given buffer be used for both vertex and pixel data?*

RESOLVED: YES. All buffers can be used with all buffer bindings, in whatever combinations the application finds useful. Consider yourself warned, however, by the following issue.

4) *May implementations make use of the target as a hint to select an appropriate memory space for the buffer?*

RESOLVED: YES, as long as such behavior is transparent to the application. Some implementations may choose, for example, that they would rather stream vertex data from AGP memory, element (index) data from video memory, and pixel data from video memory. In fact, one can imagine arbitrarily complicated heuristics for selecting the memory space, based on factors such as the target, the "usage" argument, and the application's observed behavior.

While it is entirely legal to create a buffer object by binding it to GL_ARRAY_BUFFER and loading it with data, then using it with the GL_PIXEL_UNPACK_BUFFER_ARB or GL_PIXEL_PACK_BUFFER_ARB binding, such behavior is liable to confuse the driver and may hurt performance. If the driver implemented the hypothetical heuristic described earlier, such a buffer might have already been located in AGP memory, and so the driver would have to choose between two bad options: relocate the buffer into video memory, or accept lower performance caused by streaming pixel data from AGP.

5) *Should all pixel path commands be supported, or just a subset of them?*

RESOLVED: ALL. While there is little reason to believe that, say, glConvolutionFilter2D would benefit from this extension, there is no reason _not_ to support it. The complete list of commands affected by this extension is listed in issues 17 and 18.

6) *Should glPixelMap and glGetPixelMap be supported?*

RESOLVED: YES. They're not really pixel path operations, but, again, there is no good reason to omit operations, and they _are_ operations that pass around big chunks of pixel-related data. If we support glPolygonStipple, surely we should support this.

7) *How does the buffer binding state push/pop?*

RESOLVED: As part of the pixel store client state. This is analogous to how the ARB_vertex_buffer_object bindings pushed/popped as part of the vertex array client state.

8) *Should NV_pixel_data_range (PDR) be used concurrently with pixel buffer objects ?*

RESOLVED: NO. While it would be possible to allocate a memory range for PDR, using a pointer into this memory range with one of the commands affected by PBOs will not work if a pixel buffer object other than zero is bound to the buffer binding point affecting the command.

Pixel buffer objects always have higher precedence than PDR.

9) *Should the INVALID_OPERATION error be generated if a pixel command would access data outside the range of the bound PBO?*

RESOLVED: YES. This requires considering the command parameters (such as width/height/depth/format/type/pointer), the current pixel store (pack/unpack) state, and the command operation itself to determine the maximum addressed byte for the pixel command.

Brian Paul strongly recommends this behavior.

This behavior should increase the reliability of using PBO and guard against programmer mistakes.

This is particularly important for glReadPixels where returning data into a region outside the PBO could cause corruption of application memory.

Such bounds checking is substantially more expensive for VBO accesses because bounds checking on a per-vertex element basis for each of multiple enabled vertex arrays prior to performing the command compromises the performance justification of VBO.

10) *If a pixel command with a bound PBO accesses data outside the range of the PBO, thereby generating a GL_INVALID_OPERATION error, can the pixel command end up being partially processed?*

RESOLVED: NO. As for all GL errors excepting GL_OUT_OF_MEMORY situations, "the command generating the error is ignored so that it has no effect on GL state or framebuffer contents."

This means implementations must determine before the pixel command is performed whether the resulting read or write operations on the bound PBO will exceed the size of the PBO.

This means an implementation is NOT allowed to detect out of bounds accesses in the middle of performing the command.

11) *How expensive is it to predetermine whether a pixel command accessing a PBO would have an out of bounds access?*

See the "Appendix on Pack/Unpack Range" to see the computations involved in computing the access limit.

Implementations can further specialize and optimize the check to make this out of bounds checking negligible for any sizable pixel payload.

12) *Should feedback and select buffers output results into a buffer object?*

RESOLVED: That might be useful for a future extension but is not appropriate for this extension. New targets (other than PIXEL_PACK_BUFFER_ARB and PIXEL_UNPACK_BUFFER_ARB) make sense.

13) *Should NV_pixel_data_range interactions be documented in this specification?*

RESOLVED: YES. Interactions with NV_pixel_data_range are important to document to facilitate developers migrating to the multi-vendor ARB_pixel_buffer_object extension. Discussion of interactions is limited to the issues and example usage sections.

Other ARB specifications follow this policy, and Jon Leech agrees with this policy.

14) *Should an INVALID_OPERATION error be generated if the offset within a pixel buffer to a datum comprising of N basic machine units is not a multiple of N?*

RESOLVED: YES. This was stated for VBOs but no error was defined if the rule was violated. Perhaps this needs to be better specified for VBO.

For PBO, it is reasonable and cheap to enforce the alignment rule. For pixel commands it means making sure the offset is evenly divisible by the component or group size in basic machine units.

```
        This check is independent of the pixel store state because the
        pixel store state is specified in terms of pixels (not basic
        machine units) so pixel store addressing cannot create an
        unaligned access as long as the base offset is aligned.

        Certain commands (specifically glPolygonStipple,
        glGetPolygonStipple, glBitmap, glCompressedTexImage1D,
        glCompressedTexImage2D, glCompressedTexImage3D,
        glCompressedTexSubImage1D, glCompressedTexSubImage2D,
        glCompressedTexSubImage3D, and glGetCompressedTexImage) are not
        affected by this error because the data accessed is addressed
        at the granularity of basic machine units.

    15) Various commands do not make explicit reference to supporting
        packing or unpacking from a pixel buffer object but rather specify
        that parameters are handled in the same manner as glDrawPixels,
        glReadPixels, or the glCompressedTexImage commands.  So do such
        commands (example: glCompressedTexSubImage2D) use pixel buffers?

        RESOLVED:  YES.  Commands that have their behavior defined based
        on commands that read or write from pixel buffers will themselves
        read or write from pixel buffers.  Relying on this reduces the
        amount of specification language to be updated.

    16) What is the complete list of commands that can unpack (read)
        pixels from the current pixel unpack buffer object?

            glBitmap
            glColorSubTable
            glColorTable
            glCompressedTexImage1D
            glCompressedTexImage2D
            glCompressedTexImage3D
            glCompressedTexSubImage1D
            glCompressedTexSubImage2D
            glCompressedTexSubImage3D
            glConvolutionFilter1D
            glConvolutionFilter2D
            glDrawPixels
            glPixelMapfv
            glPixelMapuiv
            glPixelMapusv
            glPolygonStipple
            glSeparableFilter2D
            glTexImage1D
            glTexImage2D
            glTexImage3D
            glTexSubImage1D
            glTexSubImage2D
            glTexSubImage3D
    17) What is the complete list of commands that can pack (write)
        pixels into the current pixel pack buffer object?

            glGetCompressedTexImage
            glGetConvolutionFilter
            glGetHistogram
            glGetMinmax
            glGetPixelMapfv
            glGetPixelMapuiv
            glGetPixelMapusv
            glGetPolygonStipple
            glGetSeparableFilter,
            glGetTexImage
            glReadPixels

    18) How does support for pixel buffer objects affect the GLX protocol?

        UNRESOLVED:  See the "GLX Protocol" section.

    19) Prior to this extension, passing zero for the data argument of
        glTexImage1D, glTexImage2D, and glTexImage3D defined a texture
        image level without supplying an image.  How does this behavior
        change with this extension?

        RESOLVED:  The "unspecified image" behavior of the glTexImage
        calls only applies when bound to a zero pixel unpack buffer
        object.

        When bound to a non-zero pixel unpack buffer object, the data
        argument to these calls is treated as an offset rather than
        a pointer so zero is a reasonable and even likely value that
        corresponds to the very beginning of the buffer object's data.

        So to create a texture image level with unspecified image data,
        you MUST bind to the zero pixel unpack buffer object.

        See the ammended language at the end of section 3.8.1.

    20) How does this extension support video frame grabbers?

        RESOLVED:  This extension extends buffer objects so they can
        operate with pixel commands, rather than just vertex array
        commands.

        We anticipate that a future extension may provide a mechanism
        for transferring video frames from video frame grabber hardware
        or vertices from motion capture hardware (or any other source
        of aquired real-time data) directly into a buffer object to
        eliminate a copy.  Ideally, such transfers would be possible
        without requiring mapping of the buffer object.  But this
        extension does not provide such functionality.

        We anticipate such functionality to involve binding a buffer
        object to a new target type, configuring a source (or sink) for
        data (video frames, motion capture vertex sets, etc.), and then
        commands to initiate data transfers to the bound buffer object.
```

21) Can this ARB extension share the same enumerants with the EXT version of this functionality?

RESOLVED: YES. The ARB extension is functionally compatible with EXT_pixel_buffer_object except that the ARB version adds additional error checks for alignment and buffer bounds checking.

The EXT behavior in the case of alignment violations and buffer bounds overflow are technically undefined. The ARB extension simply defines the EXT extension's undefined behavior to be an OpenGL error.

Using the same enumerants with firmed up error checking (that would otherwise indicate buggy usage) is preferable to two sets of enumerants where the older EXT set simply allows sloppy usage.

22) The expected usage parameters (GL_STREAM_DRAW, etc.) for glBufferData are not clearly specified. How can they be improved?

RESOLVED: To improve the clarity, replace the phrase "specified once" with "specified once per repetition of the usage pattern" so that it is clear for the STREAM_* usage modes (and the STATIC_* usage modes too, just much less frequently) that the repeated specification is part of a pattern and it is expected that the buffer can be, and will be for the STREAM_* usage patterns, specified again after being used and this is likely to repeat.

Additionally, the *_COPY and *_DRAW usage patterns can source the data with "a GL drawing command" but also with image specification commands so change this phrase to "a GL drawing or image specification command."

23) Is this the "right" way to expose render-to-vertex-array?

DISCUSSION: You can use this extension to render an image into a framebuffer, copy the pixels into a buffer object with glReadPixels, and then configure vertex arrays to source the pixel data as vertex attributes. This necessarily involves a copy from the framebuffer to the buffer object. Future extensions may provide mechanisms for copy-free render-to-vertex-array capabilities but that is not a design goal of this extension.

New Procedures and Functions

None.

New Tokens

Accepted by the <target> parameters of BindBuffer, BufferData, BufferSubData, MapBuffer, UnmapBuffer, GetBufferSubData, GetBufferParameteriv, and GetBufferPointerv:

```
    PIXEL_PACK_BUFFER_ARB                       0x88EB
    PIXEL_UNPACK_BUFFER_ARB                     0x88EC
```

Accepted by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

```
    PIXEL_PACK_BUFFER_BINDING_ARB               0x88ED
    PIXEL_UNPACK_BUFFER_BINDING_ARB             0x88EF
```

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

None

-- Section 2.9 "Buffer Objects"

Replace the first two paragraphs with:

"The vertex data arrays described in section 2.8 are stored in client memory. It is sometimes desirable to store frequently accessed client data, such as vertex array and pixel data, in high-performance server memory. GL buffer objects provide a mechanism for clients to use to allocate, initialize, and access such memory."

The name space for buffer objects is the unsigned integer, with zero reserved for the GL. A buffer object is created by binding an unused name to a buffer target. A buffer object is bound by calling

```
   void BindBuffer(enum target, uint buffer);
```

/target/ must be one of ARRAY_BUFFER, ELEMENT_ARRAY_BUFFER, PIXEL_UNPACK_BUFFER_ARB, or PIXEL_PACK_BUFFER_ARB. The ARRAY_BUFFER target is discussed in section 2.9.1 The ELEMENT_ARRAY_BUFFER target is discussed in section 2.9.2. The PIXEL_UNPACK_BUFFER_ARB and PIXEL_PACK_BUFFER_ARB targets are discussed later in sections 3.6, 4.3.2, and 6.1. If the buffer object named /buffer/ has not been previously bound or has been deleted since the last binding, the GL creates a new state vector, initialized with a zero-sized memory buffer and comprising the state values listed in table 2.6."

Replace the 5th paragraph with:

"Initially, each buffer object target is bound to zero. There is no buffer object corresponding to the name zero so client attempts to modify or query buffer object state for a target bound to zero generate an INVALID_OPERATION error."

Replace the phrase listing the valid targets for BufferData in the 9th paragraph with:

"with target set to one of ARRAY_BUFFER, ELEMENT_ARRAY_BUFFER, PIXEL_UNPACK_BUFFER_ARB, or PIXEL_PACK_BUFFER_ARB,"

In the 10th paragraph describing buffer object usage modes, replace the phrase "specified once" with "specified once per repetition of the usage pattern" for the STREAM_* and STATIC_* usage values.

Also in the 10th paragraph describing buffer object usage modes, replace the phrases "of a GL drawing command." and "for GL drawing commands." with "for GL drawing and image specification commands." for the *_DRAW and *_COPY usage values.

Replace the phrase listing the valid targets for BufferSubData in the 15th paragraph with:

"with target set to one of ARRAY_BUFFER, ELEMENT_ARRAY_BUFFER, PIXEL_UNPACK_BUFFER_ARB, or PIXEL_PACK_BUFFER_ARB."

Replace the phrase listing the valid targets for MapBuffer in the 16th paragraph with:

"with target set to one of ARRAY_BUFFER, ELEMENT_ARRAY_BUFFER, PIXEL_UNPACK_BUFFER_ARB, or PIXEL_PACK_BUFFER_ARB."

Replace the phrase listing the valid targets for UnmapBuffer in the 21st paragraph with:

"with target set to one of ARRAY_BUFFER, ELEMENT_ARRAY_BUFFER, PIXEL_UNPACK_BUFFER_ARB, or PIXEL_PACK_BUFFER_ARB."

-- Section 2.9.2 "Array Indices in Buffer Objects"

Delete the 3rd paragraph that explains how the ELEMENT_ARRAY_BUFFER target is acceptable for the commands specified in section 2.9. The updated section 2.9 language already says this.

-- NEW Section 2.9.3 "Buffer Object Required State"

"The state required to support buffer objects consists of binding names for the array buffer, element buffer, pixel unpack buffer, and pixel pack buffer. Additionally, each vertex array has an associated binding so there is a buffer object binding for each of the vertex array, normal array, color array, index array, multiple texture coordinate arrays, edge flag array, secondary color array, fog coordinate array, and vertex attribute arrays. The initial values for all buffer object bindings is zero.

The state of each buffer object consists of a buffer size in basic machine units, a usage parameter, an access parameter, a mapped boolean, a pointer to the mapped buffer (NULL if unmapped), and the sized array of basic machine units for the buffer data."

Additions to Chapter 3 of the 1.2.1 Specification (Rasterization)

-- Section 3.6 "Pixel Rectangles"

Replace the 1st sentence in the 2nd paragraph:

"A number of parameters control the encoding of pixels in buffer object or client memory (for reading and writing) and how pixels are processed before being placed in or after being read from the framebuffer (for reading, writing, and copying)."

-- RENAME Section 3.6.1 "Pixel Storage Modes and Pixel Buffer Objects"

Add to the end of the section:

"In addition to storing pixel data in client memory, pixel data may also be stored in buffer objects (described in section 2.9). The current pixel unpack and pack buffer objects are designated by the PIXEL_UNPACK_BUFFER_ARB and PIXEL_PACK_BUFFER_ARB targets respectively.

Initially, zero is bound for the PIXEL_UNPACK_BUFFER_ARB, indicating that image specification commands such as DrawPixels source their pixels from client memory pointer parameters. However, if a non-zero buffer object is bound as the current pixel unpack buffer, then the pointer parameter is treated as an offset into the designated buffer object."

-- Section 3.6.3 "Pixel Transfer Modes", page 116.

Replace the last phrase in the 2nd paragraph with:

"and /values/ refers to an array of size map values."

[values is no longer necessarily a pointer.]

Add the following paragraph after the third paragraph:

"If a pixel unpack buffer is bound (as indicated by a non-zero value of PIXEL_UNPACK_BUFFER_BINDING_ARB), /values/ is an offset into the pixel unpack buffer; otherwise, /values/ is a pointer to a block client memory. All pixel storage and pixel transfer modes are ignored when specifying a pixel map. n machine units are read where n is the /size/ of the pixel map times the size of a float, uint, or ushort datum in basic machine units, depending on the respective PixelMap version. If a pixel unpack buffer object is bound and data+n is greater than the size of the pixel buffer, INVALID_OPERATION results. If a pixel unpack buffer object is bound and /values/ is not evenly divisible into the number of basic machine units needed to store in memory a float, uint, or ushort datum depending on their respective PixelMap version, INVALID_OPERATION results."

-- Section 3.6.4 "Rasterization of Pixel Rectangles", page 126.

Change the 1st sentence of the 1st paragraph to read:

"The process of drawing pixels encoded in buffer objects or client memory is diagrammed in figure 3.7."

Change the 4th sentence of the 2nd paragraph to read:

"/data/ refers to the data to be drawn."

[data is no longer necessarily a pointer.]

Change the initial phrase in the 1st sentence of the 1st paragraph after "Unpacking" to read:

"Data are taken from the currently bound pixel unpack buffer or client memory as a sequence of..."

Insert this paragraph after the 1st paragraph after "Unpacking":

"If a pixel unpack buffer is bound (as indicated by a non-zero value of PIXEL_UNPACK_BUFFER_BINDING_ARB), /data/ is an offset into the pixel unpack buffer and the pixels are unpacked from the buffer relative to this offset; otherwise, /data/ is a pointer to a block client memory and the pixels are unpacked from the client memory relative to the pointer. If a pixel unpack buffer object is bound and unpacking the pixel data according to the process described below would access memory beyond the size of the pixel unpack buffer's memory size, INVALID_OPERATION results. If a pixel unpack buffer object is bound and /data/ is not evenly divisible into the number of basic machine units needed to store in memory the corresponding GL data type from table 3.5 for the /type/ parameter, INVALID_OPERATION results."

-- Section 3.8.1 "Texture Image Specification", page 150.

Replace the last phrase in the 2nd to last sentence in the 1st paragraph with:

"and a reference to the image data in the currently bound pixel unpack buffer or client memory."

Replace the 1st sentence in the 13th paragraph with:

"The image itself (referred to by /data/) is a sequence of groups of values."

Replace the last paragraph with:

"If the data argument of TexImage1D, TexImage2D, or TexImage3D is a null pointer (a zero-valued pointer in the C implementation) and the pixel unpack buffer object is zero, a one-, two-, or three-dimensional texture array is created with the specified target, level, internalformat, width, height, and depth border, but with unspecified image contents. In this case no pixel values are access in client memory, and no pixel processing is performed. Errors are generated, however, exactly as though the data pointer were valid. Otherwise if the pixel unpack buffer object is non-zero, the data argument is treatedly normally to refer to the beginning of the pixel unpack buffer object's data."

-- Section 3.8.3 "Compressed Texture Images", page 163.

Replace the 3rd sentence of the 2nd paragraph with:

"/data/ refers to compressed image data stored in the compressed image format corresponding to internalformat. If a pixel unpack buffer is bound (as indicated by a non-zero value of PIXEL_UNPACK_BUFFER_BINDING_ARB), /data/ is an offset into the pixel unpack buffer and the compressed data is read from the buffer relative to this offset; otherwise, /data/ is a pointer to a block client memory and the compressed data is read from the client memory relative to the pointer."

Replace the 2nd sentence in the 3rd paragraph with:

"Compressed texture images are treated as an array of /imageSize/ ubytes relative to /data/. If a pixel unpack buffer object is bound and data+imageSize is greater than the size of the pixel buffer, INVALID_OPERATION results."

Additions to Chapter 4 of the 1.2.1 Specification (Per-Fragment Operations and the Frame Buffer)

-- Section 4.3.2 "Reading Pixels", page 219.

Replace 1st sentence of the 1st paragraph with:

"The method for reading pixels from the framebuffer and placing them in pixel pack buffer or client memory is diagrammed in figure 4.2."

Add this paragraph after the 1st paragraph:

"Initially, zero is bound for the PIXEL_PACK_BUFFER_ARB, indicating that image read and query commands such as ReadPixels return pixels results into client memory pointer parameters. However, if a non-zero buffer object is bound as the current pixel pack buffer, then the pointer parameter is treated as an offset into the designated buffer object."

Rename "Placement in Client Memory" to "Placement in Pixel Pack Buffer or Client Memory".

Insert this paragraph after the newly renamed "Placement in Pixel Pack Buffer or Client Memory" heading:

"If a pixel pack buffer is bound (as indicated by a non-zero value of PIXEL_PACK_BUFFER_BINDING_ARB), /data/ is an offset into the pixel pack buffer and the pixels are packed into the buffer relative to this offset; otherwise, /data/ is a pointer to a block client memory and the pixels are packed into the client memory relative to the pointer. If a pixel pack buffer object is bound and packing the pixel data according to the pixel pack storage state would access memory beyond the size of the pixel pack buffer's memory size, INVALID_OPERATION results. If a pixel pack buffer object is bound and /data/ is not evenly divisible into the number of basic machine units needed to store in memory the corresponding GL data type from table 3.5 for the /type/ parameter, INVALID_OPERATION results."

Additions to Chapter 5 of the 1.2.1 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.2.1 Specification (State and State Requests)

-- Section 6.1.3 "Enumerated Queries".

After the sentence in the last paragraph describing GetPixelMap, add:

"The GetPixelMapfv, GetPixelMapuiv, and GetPixelMapusv commands write all the values in the named pixel map to /data/. If a pixel pack buffer is bound (as indicated by a non-zero value of PIXEL_PACK_BUFFER_BINDING_ARB), /data/ is an offset into the pixel pack buffer; otherwise, /data/ is a pointer to a block client memory. All pixel storage and pixel transfer modes are ignored when returning a pixel map. n machine units are written where n is the size of the pixel map times the size of FLOAT, UNSIGNED_INT, or UNSIGNED_SHORT respectively in basic machine units. If a pixel pack buffer object is bound and data+n is greater than the size of the pixel buffer, generate INVALID_OPERATION."

-- Section 6.1.4 "Texture Queries".

Remove the mention of img in the last phrase in the last sentence of the 1st paragraph so the sentence reads:

"lod is a level-of-detail number, format is a pixel format from table 3.6, and type is a pixel type from table 3.5."

Replace the 3rd sentence of the 2nd paragraph with:

"These groups are then packed and placed in client or pixel buffer object memory. If a pixel pack buffer is bound (as indicated by a non-zero value of PIXEL_PACK_BUFFER_BINDING_ARB), /img/ is an offset into the pixel pack buffer; otherwise, /img/ is a pointer to a block client memory."

Add to the end of the 4th paragraph:

"If a pixel pack buffer object is bound and packing the texture image into the buffer's memory would exceed the size of the buffer, generate INVALID_OPERATION."

Replace the 2nd sentence of the 5th paragraph with:

"When called, GetCompressedTexImage writes n ubytes of compressed image data to the pixel pack buffer or client memory pointed to by ptr, where n is the texture image's TEXTURE_COMPRESSED_IMAGE_SIZE value.

Add to the end of the 6th paragraph:

"If a pixel pack buffer object is bound and ptr+n is greater than the size of the buffer, generate INVALID_OPERATION."

-- Section 6.1.5 "Stipple Query".

"The pattern is packed into client or pixel pack buffer memory according to the procedures given in section 4.3.2 for ReadPixels; ..."

-- Section 6.1.7 "Color Table Query".

"The one-dimensional color table image is returned to client or pixel pack buffer memory starting at table."

-- Section 6.1.8 "Convolution Query".

"The one-dimensional or two-dimensional image is returned to client or pixel pack buffer memory starting at image."

"The row and column images are returned to client or pixel pack buffer memory starting at row and column respectively."

-- Section 6.1.9 "Histogram Query".

"The one-dimensional histogram table image is returned to client or pixel pack buffer memory starting at values."

-- Section 6.1.10 "Minmax Query".

"A one-dimensional image of width 2 is returned to client or pixel pack buffer memory starting at values."

-- Section 6.1.13 "Buffer Object Queries".

Change the 2nd sentence of the 2nd paragraph to read:

"target is ARRAY_BUFFER, ELEMENT_ARRAY_BUFFER, PIXEL_PACK_BUFFER_ARB, or PIXEL_UNPACK_BUFFER_ARB."

Change the last phrase in the 1st sentence of the 4th paragraph to:

"with target set to ARRAY_BUFFER, ELMENT_ARRAY_BUFFER, PIXEL_PACK_BUFFER_ARB, or PIXEL_UNPACK_BUFFER_ARB and pname set to BUFFER_MAP_POINTER."

GLX Protocol

XXX still-in-progress

(ARB_vertex_buffer_object has similar issues and lacks specified GLX protocol for its functionality. This discussion just addresses the issues created by pixel buffer objects, not buffer objects in general.)

Pixel buffers, like texture objects and display lists, are server-side state.

Prior to pixel buffer objects, pixel storage state for image packing and unpacking was considered client-side state. However, pixel buffers create the new situation where the server performs packing and unpacking into server-side pixel buffers.

The GLX protocol is designed so that the amount of unpacking done by the client is parameterized with the request. In other words, the client can do as much unpacking as it wants, and then tell the server what unpacking remains to be done by sending the appropriate pixel storage parameters along with the image.

This means the GLX protocol for rendering commands involving pixel data includes pixel store state for unpacking.

This means, in theory, the existing protocol for rendering commands with pixel data is sufficient for manipulating pixel buffers.
A command (for example, glDrawPixels) could build a protocol request containing the current pixel unpack state and specify zero bytes of image payload when operating on a pixel buffer object.

In practice, while this addresses command requiring unpacking of pixel data, commands that require packing of pixel data (for example, glReadPixels) to return pixel data do not have protocol fields for pixel store pack state.

Fortunately, the GLX protocol, through foresight or oversight, has GLX protocol and non-rendering command opcodes (109 and 110) assigned for glPixelStoref and glPixelStorei respectively.

It is better to use the existing protocol to send glPixelStorei and glPixelStoref GLX commands. This solves the problem of server-side pixel state the same way for both pack and unpack state. It may also allow implementations to minimize validation overhead for pixel commands because the pixel store modes are stateful rather than being parameters sent with every pixel command.

To avoid creating useless protocol overhead for applications not using pixel buffer objects, and hence not requiring server-side knowledge of pixel store state, the GLX client library is free to defer pixel store commands until just prior to pixel commands operating on pixel buffer objects that require server-side pixel store state.

There is no GLX protocol however for glPushClientAttrib and glPopClientAttrib. New protocol should be specified for these commands. These commands are also needed for vertex buffer objects because the vertex array state becomes server-side.

When bound to an pixel unpack buffer object, the pixel payload for a non-reply pixel command (for example, glTexImage2D) can be ignored. In fact, GLX client implementations are expected to send zero bytes of pixel payload in this case.

When bound to a pixel pack buffer object, the reply for pixel commands that return pixel data (for example, glReadPixels) is not required since the pixel data is actually transferred to the server-side pixel pack buffer object. Indeed, forcing an unnecessary reply would hinder the performance advantages of using pixel buffer objects

```
    Therefore, protocol for "no reply" version of the following commands
    is specified:

        GetCompressedTexImage_noreply
        GetConvolutionFilter_noreply
        GetHistogram_noreply
        GetMinmax_noreply
        GetPixelMapfv_noreply
        GetPixelMapuiv_noreply
        GetPixelMapusv_noreply
        GetPolygonStipple_noreply
        GetSeparableFilter,_noreply
        GetTexImage_noreply
        ReadPixels_noreply

    If a "no reply" command is sent when the current pixel pack
    buffer object binding is zero, a GLXBadContextState error should
    be generated by the server.
```

Errors

```
    INVALID_ENUM is generated if the <target> parameter of
    BindBuffer, BufferData, BufferSubData, MapBuffer, UnmapBuffer,
    GetBufferSubData, GetBufferParameteriv, or GetBufferPointerv is not
    one of ARRAY_BUFFER, ELEMENT_ARRAY_BUFFER, PIXEL_PACK_BUFFER_ARB,
    or PIXEL_UNPACK_BUFFER_ARB.

    INVALID_OPERATION is generated if Bitmap, ColorSubTable, ColorTable,
    CompressedTexImage1D, CompressedTexImage2D, CompressedTexImage3D,
    CompressedTexSubImage1D, CompressedTexSubImage2D,
    CompressedTexSubImage3D, ConvolutionFilter1D, ConvolutionFilter2D,
    DrawPixels, PixelMapfv, PixelMapuiv, PixelMapusv, PolygonStipple,
    SeparableFilter2D, TexImage1D, TexImage2D, TexImage3D, TexSubImage1D,
    TexSubImage2D, or TexSubImage3D would unpack (read) data from the
    currently bound PIXEL_UNPACK_BUFFER_ARB buffer object such that
    the memory reads required for the command would exceed the memory
    (data store) size of the buffer object.

    INVALID_OPERATION is generated if GetColorTable,
    GetCompressedTexImage, GetConvolutionFilter, GetHistogram, GetMinmax,
    GetPixelMapfv, GetPixelMapuiv, GetPixelMapusv, GetPolygonStipple,
    GetSeparableFilter, GetTexImage, or ReadPixels would pack (write) data
    to the currently bound PIXEL_PACK_BUFFER_ARB buffer object such that
    the memory writes required for the command would exceed the memory
    (data store) size of the buffer object.

    INVALID_OPERATION is generated by GetColorTable, GetConvolutionFilter,
    GetHistogram, GetMinmax, GetSeparableFilter, GetTexImage and ReadPixels
    if the current PIXEL_PACK_BUFFER_BINDING_ARB value is non-zero and the
    table/image/values/span/img/data parameter is not evenly divisible
    into the number of basic machine units needed to store in memory a
    datum indicated by the type parameter.

    INVALID_OPERATION is generated by ColorTable, ColorSubTable,
    ConvolutionFilter2D, ConvolutionFilter1D, SeparableFilter2D,
    TexImage1D, TexImage2D, TexImage3D, TexSubImage1D,
    TexSubImage2D, TexSubImage3D, and DrawPixels if the current
```

PIXEL_UNPACK_BUFFER_BINDING_ARB value is non-zero and the data parameter is not evenly divisible into the number of basic machine units needed to store in memory a datum indicated by the type parameter.

INVALID_OPERATION is generated by GetPixelMapfv if the current PIXEL_PACK_BUFFER_BINDING_ARB value is non-zero and the data parameter is not evenly divisible into the number of basic machine units needed to store in memory a float datum.

INVALID_OPERATION is generated by GetPixelMapuiv if the current PIXEL_PACK_BUFFER_BINDING_ARB value is non-zero and the data parameter is not evenly divisible into the number of basic machine units needed to store in memory a uint datum.

INVALID_OPERATION is generated by GetPixelMapusv if the current PIXEL_PACK_BUFFER_BINDING_ARB value is non-zero and the data parameter is not evenly divisible into the number of basic machine units needed to store in memory a ushort datum.

INVALID_OPERATION is generated by PixelMapfv if the current PIXEL_UNPACK_BUFFER_BINDING_ARB value is non-zero and the data parameter is not evenly divisible into the number of basic machine units needed to store in memory a float datum.

INVALID_OPERATION is generated by PixelMapuiv if the current PIXEL_UNPACK_BUFFER_BINDING_ARB value is non-zero and the data parameter is not evenly divisible into the number of basic machine units needed to store in memory a uint datum.

INVALID_OPERATION is generated by PixelMapusv if the current PIXEL_UNPACK_BUFFER_BINDING_ARB value is non-zero and the data parameter is not evenly divisible into the number of basic machine units needed to store in memory a ushort datum.

Dependencies on EXT_pixel_buffer_object

When this extension is supported, the EXT_pixel_buffer_object functionality adopts the tighter alignment and buffer bounds overflow error generation behavior of ARB_pixel_buffer_object (previously, EXT_pixel_buffer_object was not explicit about what happened in these situations). This is because the two extensions share the same enumerants.

Dependencies on NV_pixel_data_range

A non-zero pixel pack buffer binding takes priority over the READ_PIXEL_DATA_RANGE_NV enable.

A non-zero pixel unpack buffer binding takes priority over the WRITE_PIXEL_DATA_RANGE_NV enable.

New State

(table 6.20, Pixels, p. 235)

Get Value	Type	Get Command	Initial Value	Sec	Attribute
PIXEL_PACK_BUFFER_BINDING_ARB	Z+	GetIntegerv	0	4.3.5	pixel-store
PIXEL_UNPACK_BUFFER_BINDING_ARB	Z+	GetIntegerv	0	6.1.13	pixel-store

New Implementation Dependent State

(none)

Usage Examples

Convenient macro definition for specifying buffer offsets:

```
#define BUFFER_OFFSET(i) ((char *)NULL + (i))
```

Example 1: Render to vertex array:

```
const int numberVertices = 100;

// Create a buffer object for a number of vertices consisting of
// 4 float values per vertex
glGenBuffers(1, vertexBuffer);
glBindBuffer(GL_PIXEL_PACK_BUFFER_ARB, vertexBuffer);
glBufferData(GL_PIXEL_PACK_BUFFER_ARB, numberVertices*4,
             NULL, GL_DYNAMIC_DRAW);

// Render vertex data into 100x1 strip of framebuffer using a
// fragment program
glBindProgram(FRAGMENT_PROGRAM_ARB, fragmentProgram);
glDrawBuffer(GL_BACK);
renderVertexData();
glBindProgramARB(FRAGMENT_PROGRAM_ARB, 0);

// Read the vertex data back from framebuffer
glReadBuffer(GL_BACK);
glReadPixels(0, 0, numberVertices, 1, GL_BGRA, GL_FLOAT,
             BUFFER_OFFSET(0));

// Change the binding point of the buffer object to
// the vertex array binding point
glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer);

glEnableClientState(VERTEX_ARRAY);
glVertexPointer(4, GL_FLOAT, 0, BUFFER_OFFSET(0));
glDrawArrays(TRIANGLE_STRIP, 0, numberVertices);
```

Example 2: Streaming textures

Streaming textures using NV_pixel_data_range:

```
    const int texWidth = 256;
    const int texHeight = 256;
    const int texsize = texWidth * texHeight * 4;
    void *pdrMemory, *texData;

    pdrMemory = glAllocateMemoryNV(texsize, 0.0, 1.0, 1.0);

    glPixelDataRangeNV(GL_WRITE_PIXEL_DATA_RANGE_NV, texsize,
                       pdrMemory);

    glEnableClientState(GL_WRITE_PIXEL_DATA_RANGE_NV);

    // Define texture level (without an image)
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, texWidth, texHeight, 0,
                 GL_BGRA, GL_UNSIGNED_BYTE, NULL);
    // Setup texture environment
    ...

    texData = getNextImage();

    while (texData) {

        memcpy(pdrMemory, texData, texsize);

        glFlushPixelDataRangeNV(GL_WRITE_PIXEL_DATA_RANGE_NV);

        glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, texWidth, texHeight,
                        GL_BGRA, GL_UNSIGNED_BYTE, pdrMemory);

        // Draw textured geometry
        glBegin(GL_QUADS);
        ...
        glEnd();

        texData = getNextImage();
    }

    glDisableClientState(GL_WRITE_PIXEL_DATA_RANGE_NV);

    glFreeMemoryNV(pdrMemory);
```

Streaming textures using pixel buffer objects:

```
    const int texWidth = 256;
    const int texHeight = 256;
    const int texsize = texWidth * texHeight * 4;
    void *pboMemory, *texData;

    // Define texture level zero (without an image); notice the
    // explicit bind to the zero pixel unpack buffer object so that
    // pass NULL for the image data leaves the texture image
    // unspecified.
    glBindBuffer(GL_PIXEL_UNPACK_BUFFER_ARB, 0);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, texWidth, texHeight, 0,
                 GL_BGRA, GL_UNSIGNED_BYTE, NULL);

    // Create and bind texture image buffer object
    glGenBuffers(1, &texBuffer);
    glBindBuffer(GL_PIXEL_UNPACK_BUFFER_ARB, texBuffer);

    // Setup texture environment
    ...

    texData = getNextImage();

    while (texData) {

        // Reset the contents of the texSize-sized buffer object
        glBufferData(GL_PIXEL_UNPACK_BUFFER_ARB, texSize, NULL,
                     GL_STREAM_DRAW);

        // Map the texture image buffer (the contents of which
        // are undefined due to the previous glBufferData)
        pboMemory = glMapBuffer(GL_PIXEL_UNPACK_BUFFER_ARB,
                                GL_WRITE_ONLY);

        // Modify (sub-)buffer data
        memcpy(pboMemory, texData, texsize);

        // Unmap the texture image buffer
        glUnmapBuffer(GL_PIXEL_UNPACK_BUFFER_ARB);

        // Update (sub-)teximage from texture image buffer
        glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, texWidth, texHeight,
                        GL_BGRA, GL_UNSIGNED_BYTE, BUFFER_OFFSET(0));

        // Draw textured geometry
        glBegin(GL_QUADS);
        ...
        glEnd();

        texData = getNextImage();
    }

    glBindBuffer(GL_PIXEL_UNPACK_BUFFER_ARB, 0);
```

Example 3: Asynchronous glReadPixels

Traditional glReadPixels:

```
const int imagewidth = 640;
const int imageheight = 480;
GLubyte readBuffer[imagewidth*imageheight*4];

// Render to framebuffer
glDrawBuffer(GL_BACK);
renderScene()

// Read image from framebuffer
glReadBuffer(GL_BACK);
glReadPixels(0, 0, imagewidth, imageheight, GL_BGRA,
             GL_UNSIGNED_BYTE, readBuffer);

// Process image when glReadPixels returns after reading the
// whole buffer
processImage(readBuffer);
```

Asynchronous glReadPixels:

```
const int imagewidth = 640;
const int imageheight = 480;
const int imageSize = imagewidth*imageheight*4;

glGenBuffers(2, imageBuffers);

glBindBuffer(GL_PIXEL_PACK_BUFFER_ARB, imageBuffers[0]);
glBufferData(GL_PIXEL_PACK_BUFFER_ARB, imageSize / 2, NULL,
             GL_STREAM_READ);

glBindBuffer(GL_PIXEL_PACK_BUFFER_ARB, imageBuffers[1]);
glBufferData(GL_PIXEL_PACK_BUFFER_ARB, imageSize / 2, NULL,
             GL_STREAM_READ);

// Render to framebuffer
glDrawBuffer(GL_BACK);
renderScene();

// Bind two different buffer objects and start the glReadPixels
// asynchronously. Each call will return directly after
// starting the DMA transfer.
glBindBuffer(GL_PIXEL_PACK_BUFFER_ARB, imageBuffers[0]);
glReadPixels(0, 0, imagewidth, imageheight/2, GL_BGRA,
             GL_UNSIGNED_BYTE, BUFFER_OFFSET(0));

glBindBuffer(GL_PIXEL_PACK_BUFFER_ARB, imageBuffers[1]);
glReadPixels(0, imageheight/2, imagewidth, imageheight/2, GL_BGRA,
             GL_UNSIGNED_BYTE, BUFFER_OFFSET(0));

// Process partial images.  Mapping the buffer waits for
// outstanding DMA transfers into the buffer to finish.
glBindBuffer(GL_PIXEL_PACK_BUFFER_ARB, imageBuffers[0]);
pboMemory1 = glMapBuffer(GL_PIXEL_PACK_BUFFER_ARB,
                         GL_READ_ONLY);
```

```
        processImage(pboMemory1);
        glBindBuffer(GL_PIXEL_PACK_BUFFER_ARB, imageBuffers[1]);
        pboMemory2 = glMapBuffer(GL_PIXEL_PACK_BUFFER_ARB,
                                 GL_READ_ONLY);
        processImage(pboMemory2);

        // Unmap the image buffers
        glBindBuffer(GL_PIXEL_PACK_BUFFER_ARB, imageBuffers[0]);
        glUnmapBuffer(GL_PIXEL_PACK_BUFFER_ARB);
        glBindBuffer(GL_PIXEL_PACK_BUFFER_ARB, imageBuffers[1]);
        glUnmapBuffer(GL_PIXEL_PACK_BUFFER_ARB);
```

Appendix on Pack/Unpack Range

The complexity of OpenGL's pixel pack/unpack state makes it difficult to express succinctly what range of a pixel buffer object will be accessed by a pixel command.

The following code, following the conventions of the SGI OpenGL Sample Implementation, returns the limit (one byte more than the maximum allowed offset into the buffer object) for the memory a pixel command will read/write.

```
    /*
    ** Compute offset limit into user's data considering all pixel
    ** store modes.  This offset limit is ONE MORE than the largest byte
    ** offset for the image.
    */
    static GLsizeiptr OffsetLimitImage3D(__GLpixelStoreMode *pixelStoreMode,
                                         GLsizei width, GLsizei height,
                                         GLsizei depth,
                                         GLenum format, GLenum type,
                                         const GLvoid *userdata,
                                         GLint skip_images)
    {
        const GLint line_length = pixelStoreMode->lineLength;
        const GLint image_height = pixelStoreMode->imageHeight;
        const GLint alignment = pixelStoreMode->alignment;
        const GLint skip_pixels = pixelStoreMode->skipPixels;
        const GLint skip_lines = pixelStoreMode->skipLines;

        GLsizeiptr offsetLimit = (GLsizeiptr) userdata;

        GLint rowsize;
        GLint padding;
        GLint imagesize;

        assert(width > 0);
        assert(height > 0);
        assert(depth > 0);

        assert(line_length >= 0);
        assert(image_height >= 0);

        assert(skip_pixels >= 0);
        assert(skip_lines >= 0);
        assert(skip_images >= 0);
        assert((alignment == 1) ||
               (alignment == 2) ||
               (alignment == 4) ||
               (alignment == 8));

        /* All formats except GL_BITMAP fall out trivially */
        if (type == GL_BITMAP) {
            const GLint groups_per_line = (line_length > 0) ?
                                          line_length : width;
            const GLint rows_per_image = (image_height > 0) ?
                                         image_height : height;

            assert(1 == __glElementsPerGroup(format, type));

            rowsize = (groups_per_line + 7) / 8;
            padding = rowsize & (alignment-1);
            if (padding) {
                rowsize += alignment - padding;
            }
            imagesize = rows_per_image * rowsize;

            offsetLimit += imagesize    * (skip_images + depth-1);
            offsetLimit += rowsize      * (skip_lines  + height-1);
            offsetLimit += (skip_pixels + width+7)/8;
        } else {
            const GLint components = __glElementsPerGroup(format, type);
            const GLint element_size = __glBytesPerElement(type);
            const GLint group_size = element_size * components;

            if (0 == (line_length | image_height | skip_pixels |
                      skip_lines | skip_pixels)) {
                // Fast path: when above pixel store modes are all zero.
                rowsize = width * group_size;
                // Default alignment is 4 so allow arbitrary alignment
                // on fast path.
                padding = rowsize & (alignment-1);
                if (padding) {
                    rowsize += alignment - padding;
                }
                imagesize = depth * height * rowsize;
                offsetLimit += imagesize;
            } else {
                // General path: when one or more non-zero pixel store modes.
                const GLint groups_per_line = (line_length > 0) ?
                                              line_length : width;
                const GLint rows_per_image = (image_height > 0) ?
                                             image_height : height;

                rowsize = groups_per_line * group_size;
                padding = rowsize & (alignment-1);
                if (padding) {
                    rowsize += alignment - padding;
                }
                imagesize = rows_per_image * rowsize;
                offsetLimit += imagesize    * (skip_images  + depth-1);
                offsetLimit += rowsize      * (skip_lines   + height-1);
                offsetLimit += group_size   * (skip_pixels  + width);
            }
        }
        return offsetLimit;
    }

    GLsizeiptr __glOffsetLimitImage3D(__GLpixelStoreMode *pixelStoreMode,
                                      GLsizei width, GLsizei height,
                                      GLsizei depth,
                                      GLenum format, GLenum type,
                                      const GLvoid *userdata)
    {
        return OffsetLimitImage3D(pixelStoreMode,
                                  width, height, depth, format, type,
                                  userdata,
                                  pixelStoreMode->skipImages);
    }

    GLsizeiptr __glOffsetLimitImage(__GLpixelStoreMode *pixelStoreMode,
                                    GLsizei width, GLsizei height,
                                    GLenum format, GLenum type,
                                    const GLvoid *userdata)
    {
        /* NOTE: Non-3D image max offset computations ignore (treat as zero)
           the unpackModes.skipImages state! */
        return OffsetLimitImage3D(pixelStoreMode,
                                  width, height, 1, format, type,
                                  userdata,
                                  0);  // Treat skipImages as zero.
    }
```

Revision History

    revision 0.3: mjk

        Numbered issues.

        Add issues 14 through 18.

        Remove all gl/GL prefix/suffixing in specification sections. Use
        gl/GL prefix/suffixing in sections other than the specification
        sections. Leaving off prefixes in non-specification sections is
        ambiguous, particularly within example source code.

        Base specification language updates on OpenGL 2.0 specification.

        Add buffer object required state section.

        Added GL_INVALID_OPERATION when an offset accessed (read or
        written) for a pixel command from/to a pixel buffer object would
        exceed the size of the buffer object.

        Added GL_INVALID_OPERATION when for misaligned offsets.

```
        Added "Appendix on Pack/Unpack Range".

        Add GLX protocol discussion.

    revision 0.4: mjk

        Fixed grammar issues from Brian Paul.

        Improved example code and fixed grammar from Nick Carter.

        Explain how a NULL data parameter to glTexImage commands works.

    revision 0.5: mjk

        Clarify that glBufferData usage modes apply to drawing _and_
        image specification commands.

    revision 0.6: mjk

        Add "streaming draw pixels" to the list of interesting approaches
        for this extension in the Overview.

        Add issue discussing the relationship of this extension to data
        aquisition hardware.

    revision 0.7: mjk

        Assign enumerant values to match the EXT_pixel_buffer_object values.

        Add issue explaining why the ARB extension shares enums with
        EXT_pixel_buffer_object.

        Apply Dale's suggestion to improve the clarity of the usage
        pattern parameters to glBufferData.

    revision 0.8 mjk

        Typo fixes from Ian Romanick and Nick Carter.

    revision 1.0 mjk

        Add issue 23 for Jeremy about render-to-vertex-array.  Move
        render-to-vertex-array justification in overview to bottom of
        the list.
```

Name

    ARB_point_parameters

Name Strings

    GL_ARB_point_parameters

Status

    Approved by the ARB, 21 June 2000.

Version

    Revision Date: March 12, 2002
    Version: 0.5

    Based on: EXT_point_parameters
               $Date: 1997/08/21 21:26:36 $ $Revision: 1.6 $

Number

    ARB Extension #14

Dependencies

    OpenGL 1.0 is required.
    ARB_multisample affects the definition of this extension.
    The extension is written against the OpenGL 1.2.1 Specification.

Overview

This extension supports additional geometric characteristics of points. It can be used to render particles or tiny light sources, commonly referred to as "Light points".

The raster brightness of a point is a function of the point area, point color, point transparency, and the response of the display's electron gun and phosphor. The point area and the point transparency are derived from the point size, currently provided with the <size> parameter of glPointSize.

The primary motivation is to allow the size of a point to be affected by distance attenuation. When distance attenuation has an effect, the final point size decreases as the distance of the point from the eye increases.

The secondary motivation is a mean to control the mapping from the point size to the raster point area and point transparency. This is done in order to increase the dynamic range of the raster brightness of points. In other words, the alpha component of a point may be decreased (and its transparency increased) as its area shrinks below a defined threshold.

This extension defines a derived point size to be closely related to point brightness. The brightness of a point is given by:

```
                    1
dist_atten(d) = -------------------
                a + b * d + c * d^2

brightness(Pe) = Brightness * dist_atten(|Pe|)
```

where 'Pe' is the point in eye coordinates, and 'Brightness' is some initial value proportional to the square of the size provided with PointSize. Here we simplify the raster brightness to be a function of the rasterized point area and point transparency.

```
            brightness(Pe)       brightness(Pe) >= Threshold_Area
area(Pe) =
            Threshold_Area       Otherwise

factor(Pe) = brightness(Pe)/Threshold_Area

alpha(Pe) = Alpha * factor(Pe)
```

where 'Alpha' comes with the point color (possibly modified by lighting).

'Threshold_Area' above is in area units. Thus, it is proportional to the square of the threshold provided by the programmer through this extension.

The new point size derivation method applies to all points, while the threshold applies to multisample points only.

IP Status

None.

Issues

* *Does point alpha modification affect the current color ?*

No.

* *Do we need a special function GetPointParameterfvARB, or get by with GetFloat ?*

GetFloat is sufficient.

* *If alpha is 0, then we could toss the point before it reaches the fragment stage.*

No. This can be achieved with enabling the alpha test with reference of 0 and function of LEQUAL.

* *Do we need a disable for applying the threshold ? The default threshold value is 1.0. It is applied even if the point size is constant.*

If the default threshold is not overridden, the area of multisample points with provided constant size of less than 1.0, is mapped to 1.0, while the alpha component is modulated accordingly, to compensate for the larger area. For multisample points this is not a problem, as there are no relevant applications yet. As mentioned above, the threshold does not apply to alias or antialias points.

The alternative is to have a disable of threshold application, and state that threshold (if not disabled) applies to non antialias points only (that is, alias and multisample points).

The behavior without an enable/disable looks fine.

* *Future extensions (to the extension)*

1. POINT_FADE_ALPHA_CLAMP_ARB

When the derived point size is larger than the threshold size defined by the POINT_FADE_THRESHOLD_SIZE_ARB parameter, it might be desired to clamp the computed alpha to a minimum value, in order to keep the point visible. In this case the formula below change:

```
factor = (derived_size/threshold)^2

                factor                  clamp <= factor
clamped_value =
                clamp                   factor < clamp

         1.0                            derived_size >= threshold
alpha *=
         clamped_value                  Otherwise
```

where clamp is defined by the POINT_FADE_ALPHA_CLAMP_ARB new parameter.

New Procedures and Functions

```
void PointParameterfARB(enum pname,
                        float param);
void PointParameterfvARB(enum pname,
                         float *params);
```

New Tokens

Accepted by the <pname> parameter of PointParameterfARB, and the <pname> of Get:

```
POINT_SIZE_MIN_ARB
POINT_SIZE_MAX_ARB
POINT_FADE_THRESHOLD_SIZE_ARB
```

    Accepted by the <pname> parameter of PointParameterfvARB, and the
    <pname> of Get:

```
        POINT_SIZE_MIN_ARB               0x8126
        POINT_SIZE_MAX_ARB               0x8127
        POINT_FADE_THRESHOLD_SIZE_ARB    0x8128
        POINT_DISTANCE_ATTENUATION_ARB   0x8129
```

Additions to Chapter 2 of the OpenGL 1.2.1 Specification (OpenGL Operation)

    None

Additions to Chapter 3 of the OpenGL 1.2.1 Specification (Rasterization)

In section 3.3, the following is inserted after the description of PointSize:

The point size is multiplied with a distance attenuation factor and clamped as follows:

```
        derived_size = Clamp(size * sqrt(dist_atten(dist)))
```

where dist_atten is specified as

```
                                 1
        dist_atten(d) = -------------------
                        a + b * d + c * d^2
```

and 'd' is the eye-coordinate distance from the eye, (0, 0, 0, 1) in eye coordinates, to the vertex.

The derived point size is clamped to a given range, and then clamped to the implementation-dependent point size range.

If multisampling is enabled, an implementation may optionally fade the point alpha (section 3.12) instead of allowing the size to go below a given threshold. In this case, the diameter of the rasterized point is

```
                   derived_size                derived_size >= threshold
        diameter =
                   threshold                   Otherwise
```

and the fade factor is computed as follows:

```
                 1                             derived_size >= threshold
        fade =
                 (derived_size/threshold)^2    Otherwise
```

The distance attenuation function coefficients, 'a', 'b', and 'c', the bounds of the clamp, and the point fade 'threshold', are specified with

```
        void PointParameterfARB( enum pname, float param );
        void PointParameterfvARB( enum pname, const float *params );
```

If <pname> is POINT_SIZE_MIN_ARB or POINT_SIZE_MAX_ARB, then <param> specifies, or <params> points to the lower or upper bound respectively on the derived point size. If the lower bound is greater than the upper bound, the resulting point size is undefined. If <pname> is POINT_DISTANCE_ATTENUATION_ARB, then <params> points to the coefficients 'a', 'b', and 'c'. If <pname> is POINT_FADE_THRESHOLD_SIZE_ARB, <param> specifies, or <params> points to the point fade threshold.

This extension doesn't change the feedback or selection behavior of points.

In section 3.11, the word "Finally" is removed from the first sentence.

Add the following after section 3.11.

Section 3.12 Multisample Point Fade

If multisampling is enabled and the rasterized fragment results from a point primitive, then the computed fade factor is applied to the fragment. In RGBA mode, the fade factor is multiplied by the fragment's alpha (A) value to yield a final alpha value. In color index mode, the fade factor has no effect.

Additions to Chapter 4 of the OpenGL 1.2.1 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the OpenGL 1.2.1 Specification (Special Functions)

None

Additions to Chapter 6 of the OpenGL 1.2.1 Specification (State and State Requests)

None

Additions to Appendix A of the OpenGL 1.2.1 Specification (Invariance)

None

Additions to the GLX / WGL / AGL Specifications

None

GLX Protocol

Two new GL rendering commands are added. The following commands are sent to the server as part of a glXRender request:

```
    PointParameterfARB
        2           8+4*n             rendering command length
        2           2065              rendering command opcode
        4           ENUM              pname
                    0x8126    n=1   POINT_SIZE_MIN_ARB
                    0x8127    n=1   POINT_SIZE_MAX_ARB
                    0x8128    n=1   POINT_FADE_THRESHOLD_SIZE_ARB
        4           FLOAT32           param

    PointParameterfvARB
        2           8+4*n             rendering command length
        2           2066              rendering command opcode
        4           ENUM              pname
                    0x8126    n=1   POINT_SIZE_MIN_ARB
                    0x8127    n=1   POINT_SIZE_MAX_ARB
                    0x8128    n=1   POINT_FADE_THRESHOLD_SIZE_ARB
                    0x8129    n=3   POINT_DISTANCE_ATTENUATION_ARB
        4*n         LISTofFLOAT32   params
```

Dependencies on ARB_multisample

If ARB_multisample is not implemented, then the references to multisample points are invalid, and should be ignored.

Errors

INVALID_ENUM is generated if PointParameterfARB parameter <pname> is not POINT_SIZE_MIN_ARB, POINT_SIZE_MAX_ARB, or POINT_FADE_THRESHOLD_SIZE_ARB.

INVALID_ENUM is generated if PointParameterfvARB parameter <pname> is not POINT_SIZE_MIN_ARB, POINT_SIZE_MAX_ARB, POINT_FADE_THRESHOLD_SIZE_ARB, or POINT_DISTANCE_ATTENUATION_ARB

INVALID_VALUE is generated when values are out of range according to:

<pname>	valid range
POINT_SIZE_MIN_ARB	>= 0
POINT_SIZE_MAX_ARB	>= 0
POINT_FADE_THRESHOLD_SIZE_ARB	>= 0

New State

(table 6.11, p. 201)

Get Value	Type	Get Command	Initial Value	Description	Sec.	Attribute
POINT_SIZE_MIN_ARB	R+	GetFloatv	0.0	Attenuated Min point size	3.3	point
POINT_SIZE_MAX_ARB	R+	GetFloatv	M	Attenuated Max point size	3.3	point
POINT_FADE_THRESHOLD_SIZE_ARB	R+	GetFloatv	1.0	Threshold for alpha attenuation	3.3	point
POINT_DISTANCE_ATTENUATION_ARB	3xR	GetFloatv	(1.0,0.0,0.0)	Attenuation coefficients	3.3	point

M is the larger of the max antialiased and non antialiased point sizes.

New Implementation Dependent State

None

Revision History

```
11/09/1999  0.1
    - First ARB draft based on the original SGI and EXT drafts.

12/07/1999  0.2
    - clarified behavior when POINT_SIZE_MIN exceeds POINT_SIZE_MAX
    - clarified when the point size is clamped to the supported range
    - removed issues from "Errors" section
    - fixed various typos
    - Updated to new extension template
    - added GLX protocol

04/20/2000  0.3
    - rewritten to fit within the context of the 1.2 specification
    - added language describing where the fade alpha is applied.
    - added language which indicates that some implementations may not
      implement POINT_FADE_THRESHOLD_SIZE_ARB

06/20/2000  0.4
    - removed alternate behavior for fade alpha, since it is optional
    - added new section describing fade alpha application

03/12/2002  0.5
    - added GLX protocol for PointParameterfARB and assigned ropcodes
```

Name

    ARB_point_sprite

Name Strings

    GL_ARB_point_sprite

IP Status

    No known IP issues.

Status

    Approved by the ARB on July 24, 2003.

Version

```
Last Modified Date:  July 22, 2003
Revision:            7
```

Number

    ARB Extension #35

Dependencies

    Written based on the wording of the OpenGL 1.4 specification.

    NV_point_sprite affects the definition of this extension.

Overview

    Applications such as particle systems have tended to use OpenGL quads
    rather than points to render their geometry, since they would like
    to use a custom-drawn texture for each particle, rather than the
    traditional OpenGL round antialiased points, and each fragment in
    a point has the same texture coordinates as every other fragment.

    Unfortunately, specifying the geometry for these quads can be
    expensive, since it quadruples the amount of geometry required, and
    may also require the application to do extra processing to compute
    the location of each vertex.

    The purpose of this extension is to allow such applications to use
    points rather than quads. When GL_POINT_SPRITE_ARB is enabled,
    the state of point antialiasing is ignored. For each texture unit,
    the app can then specify whether to replace the existing texture
    coordinates with point sprite texture coordinates, which are
    interpolated across the point.

Issues

    *   *Should this spec say that point sprites get converted into quads?*

        RESOLVED: No, this would make the spec much uglier, because then
        we'd have to say that polygon smooth and stipple get turned off,

etc. Better to provide a formula for computing the texture
coordinates and leave them as points.

* *How are point sprite texture coordinates computed?*

RESOLVED: They move smoothly as the point moves around on the
screen, even though the pixels touched by the point do not. The
exact formula is given in the spec below.

A point sprite can be thought of as a quad whose upper-left corner
has (s,t) texture coordinates of (0,0) and whose lower-right
corner has texture coordinates of (1,1), as illustrated in
the following figure. In the figure "P" is the center of
the point sprite, and "O" is the origin (0,0) of the window
coordinate system. Note that the y window coordinate increases
from bottom-to-top but the t texture coordinate of point sprites
increases from top-to-bottom.

```
      ^
    +y| (0,0)
      |   +-----+
      |   |     |
      |   |  P  |
      |   |     |
      |   +-----+
      |       (1,1)
      |              +x
      O--------------->
```

Applications using a single texture for both point sprites and
other geometry need to account for the fixed coordinate mapping
of point sprites.

* *Is the ARB specification different from the NV version?*

RESOLVED: Yes. The point sprite R mode has been removed.
The wording has also been updated to reflect version 1.4 of the
core OpenGL specification however. The enumerant values are
unchanged.

* *How do point sizes for point sprites work?*

RESOLVED: This specification treats point sprite sizes like
antialiased point sizes, but with more leniency. Implementations
may choose to not clamp the point size to the antialiased point
size range. The set of point sprite sizes available must be
a superset of the antialiased point sizes. However, whereas
antialiased point sizes are all evenly spaced by the point size
granularity, point sprites can have an arbitrary set of sizes.
This lets implementations use, e.g., floating-point sizes.

* *Should there be a way to query the list of supported point sprite
sizes?*

RESOLVED: No. If an implementation were to use, say, a single-
precision IEEE float to represent point sizes, the list would
be rather long.

* *Do mipmaps apply to point sprites?*

  RESOLVED: Yes. They are similar to quads in this respect.

* *What of this extension's state is per-texture unit and what of this extension's state is state is global?*

  RESOLVED: The GL_POINT_SPRITE_ARB enable is global.
  The COORD_REPLACE_ARB state is per-texture unit (state set by TexEnv is per-texture unit).

* *Should there be a global on/off switch for point sprites, or should the per-unit enable imply that switch?*

  RESOLVED: There is a global switch to turn it on and off. This is probably more convenient for both driver and app, and it simplifies the spec.

* *What should the TexEnv mode for point sprites be called?*

  RESOLVED: COORD_REPLACE_ARB.

* *What is the interaction with multisample points, which are round?*

  RESOLVED: Point sprites are rasterized as squares, even in multisample mode. Leaving them as round points would make the feature useless.

* *How does the point sprite extension interact with fragment program extensions (ARB_fragment_program, NV_fragment_program, etc)?*

  RESOLVED: The primary issue is how the interpolated texture coordinate set appears when fragment attribute variables (ARB terminology) or fragment program attribute registers (NV terminology) are accessed.

  When point sprite is enabled and the GL_COORD_REPLACE_ARB state for a given texture unit is GL_TRUE, the texture coordinate set for that texture unit is (s,t,0,1) where the point sprite-overridden s and t are described in the amended Section 3.3 below. The important point is that r and q are forced to 0 and 1, respectively.

  For fragment program extensions, r and q correspond to the z and w components of the respective fragment attribute.

* *How does this extension interact with PolygonMode?*

  RESOLVED: If a polygon is rendered in point mode and POINT_SPRITE_ARB is enabled, its vertices will be rendered as point sprites.

* *How does this extension interact with the point size attenuation functionality in ARB_point_parameters and OpenGL 1.4?*

  RESOLVED: Point sprites sizes are attenuated just like the sizes of non-sprite points.

* *What push/pop attribute bits control the state of this extension?*

  RESOLVED: POINT_BIT for all the state. Also ENABLE_BIT for the POINT_SPRITE_ARB enable.

* *How are point sprites clipped?*

  RESOLVED: Point sprites are transformed as points, and standard point clipping operations are performed. This can cause point sprites that move off the edge of the screen to disappear abruptly, in the same way that regular points do. As with any other primitive, standard per-fragment clipping operations (scissoring, window ownership test) still apply.

New Procedures and Functions

None

New Tokens

Accepted by the <cap> parameter of Enable, Disable, and IsEnabled, by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev, and by the <target> parameter of TexEnvi, TexEnviv, TexEnvf, TexEnvfv, GetTexEnviv, and GetTexEnvfv:

```
    POINT_SPRITE_ARB                               0x8861
```

When the <target> parameter of TexEnvf, TexEnvfv, TexEnvi, TexEnviv, GetTexEnvfv, or GetTexEnviv is POINT_SPRITE_ARB, then the value of <pname> may be:

```
    COORD_REPLACE_ARB                              0x8862
```

When the <target> and <pname> parameters of TexEnvf, TexEnvfv, TexEnvi, or TexEnviv are POINT_SPRITE_ARB and COORD_REPLACE_ARB respectively, then the value of <param> or the value pointed to by <params> may be:

```
    FALSE
    TRUE
```

Additions to Chapter 2 of the OpenGL 1.4 Specification (OpenGL Operation)

None.

Additions to Chapter 3 of the OpenGL 1.4 Specification (Rasterization)

Insert the following paragraphs after the second paragraph of section 3.3 (page 66):

"Point sprites are enabled or disabled by calling Enable or Disable with the symbolic constant POINT_SPRITE_ARB. The default state is for point sprites to be disabled. When point sprites are enabled, the state of the point antialiasing enable is ignored.

The point sprite texture coordinate replacement mode is set with one of the commands

```
  void TexEnv{if}(enum target, enum pname, T param)
  void TexEnv{if}v(enum target, enum pname, const T *params)
```

where target is POINT_SPRITE_ARB and pname is COORD_REPLACE_ARB. The possible values for param are FALSE and TRUE. The default value for each texture unit is for point sprite texture coordinate replacement to be disabled."

Replace the first two sentences of the second paragraph of section 3.3.1 (page 67) with the following:

"The effect of a point width other than 1.0 depends on the state of point antialiasing and point sprites. If antialiasing and point sprites are disabled, ..."

Replace the first sentences of the fourth paragraph of section 3.3.1 (page 68) with the following:

"If antialiasing is enabled and point sprites are disabled, ..."

Insert the following paragraphs at the end of section 3.3.1 (page 70):

"When point sprites are enabled, then point rasterization produces a fragment for each framebuffer pixel whose center lies inside a square centered at the point's (x_w, y_w), with side length equal to the current point size.

All fragments produced in rasterizing a point sprite are assigned the same associated data, which are those of the vertex corresponding to the point, with texture coordinates s, t, and r replaced with s/q, t/q, and r/q, respectively. If q is less than or equal to zero, the results are undefined. However, for each texture unit where COORD_REPLACE_ARB is TRUE, these texture coordinates are replaced with point sprite texture coordinates. The s coordinate varies from 0 to 1 across the point horizontally left-to-right, while the t coordinate varies from 0 to 1 vertically top-to-bottom. The r and q coordinates are replaced with the constants 0 and 1, respectively.

The following formula is used to evaluate the s and t coordinates:

```
  s = 1/2 + (x_f + 1/2 - x_w) / size
  t = 1/2 - (y_f + 1/2 - y_w) / size
```

where size is the point's size, x_f and y_f are the (integral) window coordinates of the fragment, and x_w and y_w are the exact, unrounded window coordinates of the vertex for the point.

The widths supported for point sprites must be a superset of those supported for antialiased points. There is no requirement that these widths must be equally spaced. If an unsupported width is requested, the nearest supported width is used instead."

Replace the text of section 3.3.2 (page 70) with the following:

"The state required to control point rasterization consists of the floating-point point width, three floating-point values specifying the minimum and maximum point size and the point fade threshold size, three floating-point values specifying the distance attenuation coefficients, a bit indicating whether or not antialiasing is enabled, a bit indicating whether or not point sprites are enabled, and a bit for the point sprite texture coordinate replacement mode for each texture unit."

Replace the text of section 3.3.3 (page 70) with the following:

"If MULTISAMPLE is enabled, and the value of SAMPLE_BUFFERS is one, then points are rasterized using the following algorithm, regardless of whether point antialiasing (POINT_SMOOTH) is enabled or disabled. Point rasterization produces a fragment for each framebuffer pixel with one or more sample points that intersect a region centered at the point's (x_w, y_w). This region is a circle having diameter equal to the current point width if POINT_SPRITE_ARB is disabled, or a square with side equal to the current point width if POINT_SPRITE_ARB is enabled. Coverage bits that correspond to sample points that intersect the region are 1, other coverage bits are 0. All data associated with each sample for the fragment are the data associated with the point being rasterized, with the exception of texture coordinates when POINT_SPRITE_ARB is enabled; these texture coordinates are computed as described in section 3.3.

Point size range and number of gradations are equivalent to those supported for antialiased points when POINT_SPRITE_ARB is disabled. The set of point sizes supported is equivalent to those for point sprites without multisample when POINT_SPRITE_ARB is enabled."

Additions to Chapter 4 of the OpenGL 1.4 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 1.4 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 1.4 Specification (State and State Requests)

None.

Interactions with NV_point_sprite

ARB_point_sprite is a functional subset of NV_point_sprite.

The only functional difference between the extensions is that NV_point_sprite provides the POINT_SPRITE_R_MODE_NV control. This mode allows applications to specify how the r texture coordinates for point sprites are replaced. The r coordinate can be replaced with the corresponding s texture coordinate ("S" mode), left unchanged ("R" mode), or replaced with the constant zero ("ZERO" mode). ARB_point_sprite always replaces r texture coordiantes of point sprites with zero.

Since ARB_point_sprite is functionally compatible with the default r mode from NV_point_sprite, the two extensions can coexist nicely. Enumerant values from NV_point_sprite are reused.

If NV_point_sprite is supported, the language describing the replacement of r coordinates for point sprites (forced to zero) is replaced with the corresponding language from NV_point_sprite (controlled by POINT_SPRITE_R_MODE_NV).

Errors

None.

New State

(table 6.12, p. 220)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
POINT_SPRITE_ARB	B	IsEnabled	False	point sprite enable	3.3	point/enable

(table 6.17, p. 225)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
COORD_REPLACE_ARB	2* x B	GetTexEnviv	False	coordinate replacement enable	3.3	point

Revision History

```
    Initially based on the NV_point_sprite specification but updated for
    OpenGL 1.4.

    Rev.    Date    Author    Changes
    ----  --------  --------  -------------------------------------------
    7     07/22/03  pbrown    Marked point parameter issue resolved.

    6     07/18/03  pbrown    Removed POINT_SPRITE_R_MODE_CONTROL.

                              Expanded on spec issue documenting the
                              inversion of the "t" texture coordinate
                              relative to the "y" window coordinate.
                              Added issues on interaction with
                              PolygonMode, clipping, and point parameters.
                              Documented interaction with NV_point_sprite.
                              Removed now unneeded point parameter
                              interaction section and GLX protocol.
```

Name

    ARB_shadow

Name Strings

    GL_ARB_shadow

Contact

    Brian Paul (brian_e_paul 'at' yahoo.com)

Status

    Complete. Approved by ARB on February 14, 2002.

Version

    Last Modified Date: 21 January 2002

Number

    ARB Extension #23

Dependencies

    OpenGL 1.1 is required.
    ARB_depth_texture is required.
    This extension is written against the OpenGL 1.3 Specification.

Overview

    This extension clarifies the GL_SGIX_shadow extension.

    This extension supports comparing the texture R coordinate to a depth
    texture value in order to produce a boolean texture value. This can
    be used to implement shadow maps.

    The extension is written in generic terms such that other texture
    comparison modes can be accommodated in the future.

IP Status

    XXX None?

Issues

(1) How is this extension different from GL_SGIX_shadow?

      - It defines GL behaviour when the currently bound texture is not
        a depth texture.
      - It specifies that R is clamped to [0,1].
      - We use the standard GL_LEQUAL and GL_GEQUAL tokens instead of
        defining new ones.
      - The result may be ALPHA, LUMINANCE or INTENSITY.
      - A bit more is said about how depth textures are sampled.
      - The extension is generalized for comparison modes.

```
    (2) Should we use GL_LEQUAL and GL_EQUAL instead of
      GL_TEXTURE_LEQUAL_R_SGIX and GL_TEXTURE_GEQUAL_R_SGIX?

      RESOLUTION: Yes.  The old tokens are misleading.  For example,
      the GL_TEXTURE_LEQUAL_R_SGIX token should really have been named
      GL_R_LEQUAL_TEXTURE_SGIX since we're comparing R <= TEXTURE.
      This extension uses the standard GL_LEQUAL and GL_GEQUAL tokens.
      Also, the original shadow spec seems to be inconsistant with
      what was really implemented in hardware.

    (3) Use TEXTURE_COMPARE_OPERATOR_ARB or TEXTURE_COMPARE_FUNC_ARB?

      RESOLVED: Use TEXTURE_COMPARE_FUNC_ARB to be more consistant with
      the conventions of glDepthFunc(), glStencilFunc(), etc which use
      the GL_LEQUAL, GL_GEQUAL, etc tokens.

    (4) Should the result of the texture comparison be a LUMINANCE,
      INTENSITY or ALPHA texel?

      RESOLVED: Allow any of them. This is controlled by
      DEPTH_TEXTURE_MODE_ARB defined in ARB_depth_texture extension.

    (5) What if TEXTURE_COMPARE_MODE_ARB is set to COMPARE_R_TO_TEXTURE
      but the the currently bound texture is not a depth texture?

      RESOLVED: If the currently bound texture is a color (or paletted
      or color index) texture then the texture unit treats it in the
      usual manner and all texture comparison logic is bypassed.

    (6) Should the R value be clamped to [0,1] before the comparison?

      RESOLUTION: Yes, that makes sense since the depth texels are in
      the range [0,1].  Note that clamping R to [0,1] really only matters
      at the values 0 and 1.

    (7) How is bilinear or trilinear filtering implemented?

      RESOLUTION: We suggest an implementation behaviour but leave the
      details up to the implementation.  Differences here amount to the
      quality and softness of shadow edges.  Specific filtering
      algorithms could be expressed via layered extensions.  We're
      intentionally vague here to avoid IP and patent issues.

    (8) Is GL_ARB_shadow the right name for this extension?

      RESOLVED: Probably.  While this extension is expressed in rather
      generic terms which may be used by future extensions, it implements
      a rather specific operation at this time.

    (9) What about GL_SGIX_shadow_ambient?

      RESOLUTION: Omit that functionality.  It can be accomplished with
      advanced texture extensions such as GL_NV_register_combiners.
      GL_SGIX_shadow_ambient usually can't be implemented with existing
      hardware so it'll be offered as GL_ARB_shadow_ambient, rather than
      burdon this extension with it.
```

New Procedures and Functions

    None

New Tokens

    Accepted by the <pname> parameter of TexParameterf, TexParameteri,
    TexParameterfv, TexParameteriv, GetTexParameterfv, and GetTexParameteriv:

TEXTURE_COMPARE_MODE_ARB	0x884C
TEXTURE_COMPARE_FUNC_ARB	0x884D

    Accepted by the <param> parameter of TexParameterf, TexParameteri,
    TexParameterfv, and TexParameteriv when the <pname> parameter is
    TEXTURE_COMPARE_MODE_ARB:

COMPARE_R_TO_TEXTURE_ARB	0x884E

Additions to Chapter 2 of the 1.3 Specification (OpenGL Operation)

    None

Additions to Chapter 3 of the 1.3 Specification (Rasterization)

    Section 3.8.4, Texture Parameters, p. 133, append table 3.19 with the
    following:

Name	Type	Legal Values
TEXTURE_COMPARE_MODE_ARB	enum	NONE, COMPARE_R_TO_TEXTURE
TEXTURE_COMPARE_FUNC_ARB	enum	LEQUAL, GEQUAL

    After section 3.8.12, Texture Environments and Texture Functions,
    p. 149, insert the following new sections (and renumber subsequent
    sections):

"3.8.13 Texture Comparison Modes

    TEXTURE_COMPARE_MODE_ARB can be used to compute the texture value
    according to a comparison function. TEXTURE_COMPARE_MODE_ARB
    specifies the comparison operands, and TEXTURE_COMPARE_FUNC_ARB
    specifies the comparison function. The format of the resulting
    texture sample is specified by the DEPTH_TEXTURE_MODE_ARB.

3.8.13.1 Depth Texture Comparison Mode

    If the currently bound texture's format is DEPTH_COMPONENT then
    TEXTURE_COMPARE_MODE_ARB, TEXTURE_COMPARE_FUNC_ARB and
    DEPTH_TEXTURE_MODE_ARB control the output of the texture unit
    as described below. However, if the currently bound texture is
    not DEPTH_COMPONENT then the texture unit operates in the normal
    manner and texture comparison is bypassed.

    Let Dt (D subscript t) be the depth texture value, in the range
    [0, 1]. Let R be the interpolated texture coordinate clamped to

```
        the range [0, 1].  Then the effective texture value Lt, It, or At
        is computed by

        if TEXTURE_COMPARE_MODE_ARB = NONE

            r = Dt

        else if TEXTURE_COMPARE_MODE_ARB = COMPARE_R_TO_TEXTURE_ARB

            if TEXTURE_COMPARE_FUNC_ARB = LEQUAL

                 { 1.0,  if R <= Dt
             r = {
                 { 0.0,  if R > Dt

            else if TEXTURE_COMPARE_FUNC_ARB = GEQUAL

                 { 1.0,  if R >= Dt
             r = {
                 { 0.0,  if R < Dt

            endif

            if DEPTH_TEXTURE_MODE_ARB = LUMINANCE

                Lt = r

            else if DEPTH_TEXTURE_MODE_ARB = INTENSITY

                It = r

            else if DEPTH_TEXTURE_MODE_ARB = ALPHA

                At = r

            endif

        endif

        If TEXTURE_MAG_FILTER is not NEAREST or TEXTURE_MIN_FILTER is
        not NEAREST or NEAREST_MIPMAP_NEAREST then r may be computed by
        comparing more than one depth texture value to the texture R
        coordinate.  The details of this are implementation-dependent
        but r should be a value in the range [0, 1] which is proportional
        to the number of comparison passes or failures.
```

Additions to Chapter 4 of the 1.3 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.3 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.3 Specification (State and State Requests)

In section 6.1.3, p. 200, insert the following after the fourth paragraph:

"The texture compare mode and texture compare function may be queried by calling GetTexParameteriv or GetTexParameterfv with <pname> set to TEXTURE_COMPARE_MODE_ARB, or TEXTURE_COMPARE_FUNC_ARB, respectively."

Additions to the GLX Specification

None

Errors

INVALID_ENUM is generated if TexParameter[if] parameter <pname> is TEXTURE_COMPARE_MODE_ARB and parameter <param> is not NONE or COMPARE_R_TO_TEXTURE.

INVALID_ENUM is generated if TexParameter[if] parameter <pname> is TEXTURE_COMPARE_FUNC_ARB and parameter <param> is not LEQUAL or GEQUAL.

New State

In table 6.16, Texture Objects, p. 224, add the following:

Get Value	Type	Get Command	Initial Value	Description	Sec.	Attribute
TEXTURE_COMPARE_MODE_ARB	Z_2	GetTexParameter[if]v	NONE	compare mode	3.8.13	texture
TEXTURE_COMPARE_FUNC_ARB	Z_2	GetTexParameter[if]v	LEQUAL	compare func	3.8.13	texture

New Implementation Dependent State

None

Revision History

```
19 March 2001
    - initial revision
20 March 2001
    - use GL_LEQUAL, GL_GEQUAL tokens
    - removed TEXTURE_COMPARE_FAIL_VALUE_ARB
16 April 2001
    - renamed TEXTURE_COMPARE_OPERATOR_ARB to TEXTURE_COMPARE_FUNC_ARB
    - replace TEXTURE_COMPARE_ARB with TEXTURE_COMPARE_MODE_ARB
22 April 2001
    - added TEXTURE_COMPARE_RESULT
23 April 2001
    - minor tweaks
22 June 2001
    - fixed grammatical errors
16 November 2001
    - Change default value of TEXTURE_COMPARE_MODE_ARB to LUMINANCE.
17 November 2001
    - Resolved issue 5
    - cleaned up new section 3.8.7.1 yet again
12 December 2001
    - rewritten against the OpenGL 1.3 spec
3 January 2002
    - fixed a typo found by Bimal
18 January 2002
    - Since depth textures can now allow ALPHA, INTENSITY, LUMINANCE mode,
      there was no need for TEXTURE_COMPARE_RESULT_ARB.
21 January 2002
    - Fixed error to be INVALID_ENUM instead of INVALID_OPERATION.
```

Name

    ARB_texture_border_clamp

Name Strings

    GL_ARB_texture_border_clamp

Status

    Complete. Approved by the ARB, 20 June 2000

Version

    1.0, 22 June 2000

Number

    ARB Extension #13

Dependencies

    OpenGL 1.0 is required.

    This extension is written against the OpenGL 1.2.1 Specification.

    This extension is based on and intended to replace
    GL_SGIS_texture_border_clamp.

Overview

    The base OpenGL provides clamping such that the texture coordinates are
    limited to exactly the range [0,1]. When a texture coordinate is clamped
    using this algorithm, the texture sampling filter straddles the edge of
    the texture image, taking 1/2 its sample values from within the texture
    image, and the other 1/2 from the texture border. It is sometimes
    desirable for a texture to be clamped to the border color, rather than to
    an average of the border and edge colors.

    This extension defines an additional texture clamping algorithm.
    CLAMP_TO_BORDER_ARB clamps texture coordinates at all mipmap levels such
    that NEAREST and LINEAR filters return only the color of the border
    texels.

IP Status

    No known IP issues.

Issues

    (1) Is this formulation correct for higher-order texture filters
        (e.g., cubic or anisotropic filters)?

       RESOLVED: No. A more appropriate formulation would clamp the texture
       coordinates in texel space.

New Procedures and Functions

None.

New Tokens

Accepted by the <param> parameter of TexParameteri and TexParameterf, and by the <params> parameter of TexParameteriv and TexParameterfv, when their <pname> parameter is TEXTURE_WRAP_S, TEXTURE_WRAP_T, or TEXTURE_WRAP_R:

```
    CLAMP_TO_BORDER_ARB                             0x812D
```

Additions to Chapter 2 of the OpenGL 1.2.1 Specification (OpenGL Operation)

None.

Additions to Chapter 3 of the OpenGL 1.2.1 Specification (Rasterization)

Modify Table 3.17, p. 124, editing only the following lines:

Name	Type	Legal Values
TEXTURE_WRAP_S	integer	CLAMP, CLAMP_TO_EDGE, REPEAT, CLAMP_TO_BORDER_ARB
TEXTURE_WRAP_T	integer	CLAMP, CLAMP_TO_EDGE, REPEAT, CLAMP_TO_BORDER_ARB
TEXTURE_WRAP_R	integer	CLAMP, CLAMP_TO_EDGE, REPEAT, CLAMP_TO_BORDER_ARB

Modify **Section 3.8.4, Texture Wrap Modes**, p.124

(add at the end of the section, p. 125)

CLAMP_TO_BORDER_ARB clamps texture coordinates at all mipmaps such that the texture filter always samples border texels for fragments whose corresponding texture coordinate is sufficiently far outside the range [0,1]. The color returned when clamping is derived only from the border texels of the texture image, or from the constant border color if the texture image does not have a border.

Texture coordinates are clamped to the range [min, max]. The minimum value is defined as

```
    min = -1 / 2N
```

where N is the size (not including borders) of the one-, two-, or three-dimensional texture image in the direction of clamping. The maximum value is defined as

```
    max = 1 - min
```

so that clamping is always symmetric about the [0,1] mapped range of a texture coordinate.

Additions to Chapter 4 of the OpenGL 1.2.1 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 1.2.1 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 1.2.1 Specification (State and State Requests)

None.

Additions to Appendix A of the OpenGL 1.2.1 Specification (Invariance)

None.

Additions to the AGL/GLX/WGL Specifications

None.

GLX Protocol

None.

Errors

None.

New State

Only the type information changes for these parameters.

(table 6.13, p. 203)

Get Value	Type	Get Command	Initial Value	Description	Sec.	Attribute
TEXTURE_WRAP_S	3+ x Z4	GetTexParameter	REPEAT	Texture wrap	3.8	texture
TEXTURE_WRAP_T	3+ x Z4	GetTexParameter	REPEAT	Texture wrap	3.8	texture
TEXTURE_WRAP_R	3+ x Z4	GetTexParameter	REPEAT	Texture wrap	3.8	texture

Revision History

```
1.0,  06/22/2000 prbrown1:   Added issue w.r.t. higher order filters.

0.2,  05/23/2000 prbrown1:   Removed dependency on SGIS_texture_filter4
                             per ARB guidelines.

0.1,  05/02/2000 prbrown1:   Initial revision -- mostly stolen from
                             GL_SGIS_texture_border_clamp.
```

Name

    ARB_texture_compression

Name Strings

    GL_ARB_texture_compression

Status

    FINAL VERSION -- APPROVED BY OPENGL ARB, 3/16/2000.

Version

```
    Final 1.03, 23 May 2000 (supersedes Final 1.0, 24 March 2000 -
                             contains a few minor fixes documented in
                             the Revision History below).
```

Number

    ARB Extension #12

Dependencies

```
    OpenGL 1.1 is required.

    This extension is written against the OpenGL 1.2.1 Specification.

    This extension is written against the GLX Extensions for OpenGL
        Specification (Version 1.3).

    Depends on GL_ARB_texture_cube_map, as cube maps may be stored in
    compressed form.
```

Overview

```
    Compressing texture images can reduce texture memory utilization and
    improve performance when rendering textured primitives.  This extension
    allows OpenGL applications to use compressed texture images by providing:

        (1) A framework upon which extensions providing specific compressed
            image formats can be built.

        (2) A set of generic compressed internal formats that allow
            applications to specify that texture images should be stored in
            compressed form without needing to code for specific compression
            formats.

    An application can define compressed texture images by providing a texture
    image stored in a specific compressed image format.  This extension does
    not define any specific compressed image formats, but it does provide the
    mechanisms necessary to enable other extensions that do.

    An application can also define compressed texture images by providing an
    uncompressed texture image but specifying a compressed internal format.
    In this case, the GL will automatically compress the texture image using
    the appropriate image format.  Compressed internal formats can either be
```

specific (as above) or generic. Generic compressed internal formats are not actual image formats, but are instead mapped into one of the specific compressed formats provided by the GL (or to an uncompressed base internal format if no appropriate compressed format is available). Generic compressed internal formats allow applications to use texture compression without needing to code to any particular compression algorithm. Generic compressed formats allow the use of texture compression across a wide range of platforms with differing compression algorithms and also allow future GL implementations to substitute improved compression methods transparently.

Compressed texture images can be obtained from the GL in uncompressed form by calling GetTexImage and in compressed form by calling GetCompressedTexImageARB. Queried compressed images can be saved and later reused by calling CompressedTexImage[123]DARB. Pre-compressed texture images do not need to be processed by the GL and should significantly improve texture loading performance relative to uncompressed images.

This extension does not define specific compressed image formats (e.g., S3TC, FXT1), nor does it provide means to encode or decode such images. To support images in a specific compressed format, a hardware vendor would:

(1) Provide a new extension defininig specific compressed <internalformat> and <format> tokens for TexImage[123]D, TexSubImage[123]D, CopyTexImage[12]D, CompressedTexImage[123]DARB, CompressedTexSubImage[123]DARB, and GetCompressedTexImageARB calls.

(2) Specify the encoding of compressed images of that specific format.

(3) Specify a method for deriving the size of compressed images of that specific format, using the <internalformat>, <width>, <height>, <depth> parameters, and (if necessary) the compressed image itself.

IP Status

No known intellectual property issues on this general extension.

Specific compression algorithms used to implement this extension (and any other specific texture compression extensions) may be protected and require licensing agreements.

Issues

(1) Should we define additional internal formats that strongly tie an underlying compression algorithm to the format?

RESOLVED: Not here. Explicit compressed formats will be provided by other extensions built on top of this one.

(2) Should we provide additional compression state that gives more control on the level/quality of compression? If so, how?

RESOLVED: Yes, as a hint. Could have also been implemented as a [0.0, 1.0] floating-point TexParameter "quality" state variable (such as the JPEG quality scale found in many apps). This control will affect only the speed (and quality) with which a driver compresses incoming images, but will not affect the compressed image format selected by the driver.

As the spec is currently formulated, the requirement that quality control not affect compression format selection could have been relaxed by loosening the invariance requirements (so that the quality control can affect the choice of internal format). The risk was the potential for subtle mipmap consistency issues if the hint changes.

(3) Most current compression algorithms handle primarily RGB and RGBA images. Does it make sense having generic compressed formats for alpha, intensity, luminance, and luminance-alpha?

RESOLVED: Yes. It is conceivable that some or all of these formats may be compressed. Implementations not having compression algorithms for these formats can simply choose not to compress and use the appropriate base internal format instead.

(4) Full GetTexImage support requires that the renderer decompress the whole image. Should this extra implementation burden be imposed on the renderer?

RESOLVED: Yes, returning the uncompressed image is a useful feature for evaluating the quality of the compressed image. A decompression engine may also be required for a number of other areas, including software rasterization.

(5) Full TexSubImage support may require that the renderer decompress portions of the image (or perhaps the whole image), do a merge, and then recompress. Even if this were done, portions of the image outside the "modified" area may also be modified due to lossy compression. Should this extra implementation burden be imposed on the renderer?

RESOLVED: No. To avoid the complications involved with modifying a compressed texture image, only the lower-left corner may be modified by TexSubImage. In addition, after calling TexSubImage, the "unmodified" portion of the image is left undefined. An INVALID_OPERATION error results from any other TexSubImage calls.

This behavior allows for the use of compressed images whose dimensions are not powers of two, which TexImage will not accept. The recommended sequence of calls for defining such images is to first call TexImage with a NULL <data> pointer and the image size parameters padded out to the next power of two, and then call CompressedTexSubImageARB or TexSubImage with <xoffset>, <yoffset>, and <zoffset> parameters of zero and the compressed data pointed to by <data>. This behavior also allows TexSubImage to be used as a light-weight replacement of TexImage, where only the image contents are modified.

Certain compressed formats may allow a wider variety of edits -- their specifications will document the restrictions under which these edits are permitted. it is impossible to document such restrictions for unknown generic formats. It is desirable to keep the behavior of generic formats and the specific formats they map to as consistent as possible.

(6) What do the return values of the component sizes (RED_BITS, GREEN_BITS, ...) give for compressed textures? Compressed proxy textures?

RESOLVED: Some behavior has to be defined. For both normal and proxy textures, we return the bit depths of an uncompressed sized image that would most closely match the quality of the compression algorithm for an "average" texture image. Since compressed image quality is highly data dependent, the actual compressed image quality may be better or worse than the renderer's best guess at the best matching sized internal format. To implement this feature in a driver, it is expected that an error analysis would be done on a set of representative images, and the resultant "equivalent bit depths" would be hardwired constants.

(7) What should GetTexLevelParameter with TEXTURE_COMPRESSED_ IMAGE_SIZE_ARB return for existing uncompressed formats? For proxy textures?

RESOLVED: For both, an INVALID_OPERATION error results. The actual image to be compressed is not available for proxies, so actually compressing the specified image is not an option.

For uncompressed internal formats, we could return the actual amount of memory taken by the texture image. Such a mechanism might be useful as a metric of "how much space does this texture image take". It's not particularly useful for an application based texture management scheme, since there is no information available indicating the amount of available memory. In addition, because of implementation-dependent hardware constraints, the amount of texture memory consumed by a texture object is not necessarily equal to the sum of the memory consumed by each of its mipmaps. The OpenGL ARB decided against adopting this behavior when this specification was approved.

(8) What about texture borders?

RESOLVED: Not a problem for generic compressed formats since a base internal format can be used if borders are not supported in the compressed image format. Borders may pose problems for specific compression extensions, and compressed textures with borders might well be disallowed by those extensions.

(9) Should certain pixel operations be disallowed for compressed texture internal formats (e.g., PixelStorage, PixelTransfer)? What about byte swapping?

RESOLVED: For uncompressed source images, all pixel storage and pixel transfer modes will be applied prior to compression. For compressed source images, all pixel storage and transfer modes will be ignored. The encoding of compressed images should be specified as a byte stream that matches the disk file format defined for the corresponding image type.

(10) Should functionality be provided to allow applications to save compressed images to disk and reuse them in subsequent runs without programming to specific formats? If so, how?

RESOLVED: Yes. This can be done without knowledge of specific compression formats in the following manner:

* Call TexImage with an uncompressed image and a generic compressed internal format. The texture image will be compressed by the GL, if possible.

* Call GetTexLevelParameteriv with a <value> of TEXTURE_COMPRESSED_ARB to determine if the GL was able to store the image in compressed form.

* Call GetTexLevelParameteriv with a <value> of TEXTURE_INTERNAL_FORMAT to determine the specific compressed image format in which the image is stored.

* Call GetTexLevelParameteriv with a <value> of TEXTURE_COMPRESSED_IMAGE_SIZE_ARB to determine the size (in ubytes) of the compressed image that will be returned by the GL. Allocate a buffer of at least this size.

* Call GetCompressedTexImageARB. The GL will write the compressed texture image into the allocated buffer.

* Save the returned compressed image to disk, along with the associated width, height, depth, border parameters and the returned values of TEXTURE_COMPRESSED_IMAGE_SIZE_ARB and TEXTURE_INTERNAL_FORMAT.

* Load the compressed image and its parameters, and call CompressedTexImage_[123]DARB to use the compressed image. The value of TEXTURE_INTERNAL_FORMAT should be used as <internalFormat> and the value of TEXTURE_COMPRESSED_IMAGE_SIZE_ARB should be used as <imageSize>.

The saved images will be valid as long as they are used on a device supporting the returned <internalFormat> parameter. If the saved images are used on a device that does not support the compressed internal format, an INVALID_ENUM error would be generated by the call to CompressedTexImage_[123]D because of the unknown format.

Note also that to reliably determine if the GL will compress an image without actually compressing it, an application need only define a proxy texture image and query TEXTURE_COMPRESSED_ARB as above.

(11) Without knowing of the compressed image format, there is no convenient way for the client-side GLX library or tracing tools to ascertain the size of a compressed texture image when sending a TexImage1D, TexImage2D, or TexImage3D packet or interpret pixel storage modes. To complicate matters further, it is possible to create both indirect (that might not understand an image format) and direct rendering contexts (that might understand an image format) on the same renderer. How should this be solved?

RESOLVED: A separate set of CompressedTexImage and CompressedTexSubImage calls has been created that allows libraries to pass compressed images along to the renderer without needing to understand their specific image formats or how to interpret pixel storage modes.

(12) Are the CompressedTexImage[123]DARB entry points really needed?

RESOLVED: Yes. To robustly support images of unknown format, specific compressed entry points are required. While the extension does not support images in a completely unspecified format (early drafts did), having a separate call means that GLX and tools such as GLS (stream encoder) do not need intimate knowledge of every compressed image format. Having separate calls also cleanly solves the problem where pixel storage and pixel transfer operations apply if and only if the source image is uncompressed.

(13) Is variable-ratio compression supported?

RESOLVED: Yes. Fixed-ratio compression is currently the predominant texture compression format, but this spec should not preclude the use of other compression schemes.

(14) Should the <imageSize> parameter be validated on CompressedTexImage calls?

RESOLVED: Yes. Enforcement overhead is generally trivial. Without enforcement, an application could specify incorrect image sizes but notice them only when run on an indirect renderer, causing portability problems. There is also a reliability issue with respect to the GLX environment -- if the compressed image size provided by the user is less than the required image size, the GLX server may run off the end of the image and access invalid memory. A size check may thus be desirable to prevent server crashes (even though that could be considered an "undefined" result).

While enforcing correct <imageSize> parameters is trivial for current compressed internal formats, it might not be reasonable on others (particular variable-ratio compression formats). For such formats, this restriction should be overridden in the spec defining the formats. The <imageSize> check was made mandatory only in the final draft approved at the March 2000 OpenGL ARB meeting.

(15) Should TexImage calls fall back to uncompressed image formats when <internalformat> is a specific compressed format but its use in combination with other parameter values passed is not supported by the renderer?

RESOLVED: Yes. Advantages: Works in exactly the same way as generic formats, meaning no extra code/error checking. Inherent limitations of TexImage on specific formats should be documented in their specs and observed by their users. One simple query can detect fallback cases. Disadvantages: Silent fallback to a format not requested by the user.

(16) Should the texture format invariance requirements disallow scanning of the image data to select a compression method? What about for a base (uncompressed) internal format?

RESOLVED: The primary issue is mipmap consistency. The 1.2.1 spec defines a set of mipmaps as consistent if all are specified using the same internal format. However, it doesn't require that all mipmaps are allocated using the same format -- the renderer is responsible for ensuring mipmap consistency if it selects different formats for different images. There is no reason to disallow scanning for base internal formats; the renderer is responsible for doing the right thing.

The selection of a specific compressed internal format is different. It must be independent of the the image data because the GL treats the texture image as though it were specified using the specific compressed internal format chosen by the renderer.

(17) Should functionality be provided to enumerate the specific compressed formats supported by the renderer? If so, how and what will it accomplish?

RESOLVED: Yes. A glGet* query is added to return the number of compressed internal formats supported by the renderer and the <internalformat> tokens for each. These tokens can subsequently be used as <internalformat> parameters for normal TexImage calls and the new CompressedTexImage calls.

Providing an internal format enumeration allows applications to weigh the suitability of the various compression methods provided to it by the renderer without needing specific knowledge of the formats. Applications can query the component sizes (see issue 6) to determine the base format and approximate precision. Applications can directly evaluate image compression quality by having the renderer generate compressed texture images (using the returned <internalformat> values) and return them in uncompressed form using GetTexImage. Applications should also be aware that the use of the internal formats returned by this query is subject to the restrictions imposed by the specification defining them. The use of proxy textures allows the application to determine if a specific set of TexImage parameters is supported for a given internal format.

The renderer should enumerate all supported compression formats EXCEPT those that operate fundamentally differently from a normal uncompressed format. For example, the DirectX DXT1 compression format is fundamentally an RGB format, but it has a "transparent" encoding where the red, green, and blue component values are forced to zero, regardless of their original (uncompressed) values. Since such formats may have caveats that must be understood before being used, they should not be enumerated by this query.

This allows for forward compatibility -- an application can exploit compression techniques provided by future renderers.

(18) Should the separate GetCompressedTexImageARB function exist, or is GetTexImage with special <format> and/or <type> parameters sufficient?

RESOLVED: Provide a separate GetCompressedTexImageARB function. The primary rationale is for GLX indirect rendering. The client GetTexImage would require information to determine if an image is uncompressed (and should be decoded using pixel storage state) or compressed (pixel storage ignored). In addition, if the image is compressed, the actual image size would be required, but the only image size that could be inferred from the GLX protocol is padded out to a multiple of four bytes. A separate call is the cleanest solution to both issues.

New Procedures and Functions

```
    void CompressedTexImage3DARB(enum target, int level,
                                 enum internalformat, sizei width,
                                 sizei height, sizei depth,
                                 int border, sizei imageSize,
                                 const void *data);
    void CompressedTexImage2DARB(enum target, int level,
                                 enum internalformat, sizei width,
                                 sizei height, int border,
                                 sizei imageSize, const void *data);
    void CompressedTexImage1DARB(enum target, int level,
                                 enum internalformat, sizei width,
                                 int border, sizei imageSize,
                                 const void *data);
    void CompressedTexSubImage3DARB(enum target, int level,
                                    int xoffset, int yoffset,
                                    int zoffset, sizei width,
                                    sizei height, sizei depth,
                                    enum format, sizei imageSize,
                                    const void *data);
    void CompressedTexSubImage2DARB(enum target, int level,
                                    int xoffset, int yoffset,
                                    sizei width, sizei height,
                                    enum format, sizei imageSize,
                                    const void *data);
    void CompressedTexSubImage1DARB(enum target, int level,
                                    int xoffset, sizei width,
                                    enum format, sizei imageSize,
                                    const void *data);
    void GetCompressedTexImageARB(enum target, int lod,
                                  void *img);
```

New Tokens

```
    Accepted by the <internalformat> parameter of TexImage1D, TexImage2D,
    TexImage3D, CopyTexImage1D, and CopyTexImage2D:

        COMPRESSED_ALPHA_ARB                            0x84E9
        COMPRESSED_LUMINANCE_ARB                        0x84EA
        COMPRESSED_LUMINANCE_ALPHA_ARB                  0x84EB
        COMPRESSED_INTENSITY_ARB                        0x84EC
        COMPRESSED_RGB_ARB                              0x84ED
        COMPRESSED_RGBA_ARB                             0x84EE

    Accepted by the <target> parameter of Hint and the <value> parameter of
    GetIntegerv, GetBooleanv, GetFloatv, and GetDoublev:

        TEXTURE_COMPRESSION_HINT_ARB                    0x84EF

    Accepted by the <value> parameter of GetTexLevelParameter:

        TEXTURE_COMPRESSED_IMAGE_SIZE_ARB               0x86A0
        TEXTURE_COMPRESSED_ARB                          0x86A1
```

Accepted by the <value> parameter of GetIntegerv, GetBooleanv, GetFloatv, and GetDoublev:

```
    NUM_COMPRESSED_TEXTURE_FORMATS_ARB              0x86A2
    COMPRESSED_TEXTURE_FORMATS_ARB                  0x86A3
```

Additions to Chapter 2 of the OpenGL 1.2.1 Specification (OpenGL Operation)

None.

Additions to Chapter 3 of the OpenGL 1.2.1 Specification (Rasterization)

Modify **Section 3.8.1, Texture Image Specification (p.113)**

(p.113, modify 3rd paragraph) <internalformat> may be specified as one of the six base internal format symbolic constants listed in table 3.15, as one of the sized internal format symbolic constants listed in table 3.16, as one of the specific compressed internal format symbolic constants listed in table 3.16.1, or as one of the six generic compressed internal format symbolic constants listed in table 3.16.2.

(p.113, add after 3rd paragraph)

The ARB_texture_compression specification provides no specific compressed internal formats but does provide a mechanism to obtain the enums for such formats provided by other specifications. If the ARB_texture_compression extension is supported, the number of specific compressed internal format symbolic constants supported by the renderer can be obtained by querying the value of NUM_COMPRESSED_TEXTURE_FORMATS_ARB. The set of specific compressed internal format symbolic constants supported by the renderer can be obtained by querying the value of COMPRESSED_TEXTURE_FORMATS_ARB. The only symbolic constants returned by this query are those suitable for general-purpose usage. The renderer will not enumerate formats with restrictions that need to be specifically understood prior to use.

Generic compressed internal formats are never used directly as the internal formats of texture images. If <internalformat> is one of the six generic compressed internal formats, its value is replaced by the symbolic constant for a specific compressed internal format of the GL's choosing with the same base internal format. If no specific compressed format is available, <internalformat> is instead replaced by the corresponding base internal format. If <internalformat> is given as or mapped to a specific compressed internal format, but the GL can not support images compressed in the chosen internal format for any reason (e.g., the compression format might not support 3D textures or borders), <internalformat> is replaced by the corresponding base internal format and the texture image will not be compressed by the GL.

(p.113, modify 4th paragraph) ... If a compressed internal format is specified, the mapping of the R, G, B, and A values to texture components is equivalent to the mapping of the corresponding base internal format's components, as specified in table 3.15. The specified image is compressed using a (possibly lossy) compression algorithm chosen by the GL.

(p.113, 5th paragraph) A GL implementation may vary its allocation of internal component resolution or compressed internal format based on any TexImage3D, TexImage2D, or TexImage1D (see below) parameter (except

<target>, but the allocation and chosen compressed image format must not be a function of any other state and cannot be changed once they are established. In addition, the choice of a compressed image format may not be affected by the <data> parameter. Allocations must be invariant; the same allocation and compressed image format must be chosen each time a texture image is specified with the same parameter values. These allocation rules also apply to proxy textures, which are described in section 3.8.7.

Add Table 3.16.1: Specific Compressed Internal Formats

Compressed Internal Format	Base Internal Format
none provided here -- defined by dependent extensions	

Add Table 3.16.2: Generic Compressed Internal Formats

Generic Compressed Internal Format	Base Internal Format
COMPRESSED_ALPHA_ARB	ALPHA
COMPRESSED_LUMINANCE_ARB	LUMINANCE
COMPRESSED_LUMINANCE_ALPHA_ARB	LUMINANCE_ALPHA
COMPRESSED_INTENSITY_ARB	INTENSITY
COMPRESSED_RGB_ARB	RGB
COMPRESSED_RGBA_ARB	RGBA

Modify **Section 3.8.2, Alternate Image Specification**

(add to end of TexSubImage discussion, p.123)

Texture images with compressed internal formats may be stored in such a way that it is not possible to edit an image with subimage commands without having to decompress and recompress the texture image being edited. Even if the image were edited in this manner, it may not be possible to preserve the contents of some of the texels outside the region being modified. To avoid these complications, the GL does not support arbitrary edits to texture images with compressed internal formats. Calling TexSubImage3D, CopyTexSubImage3D, TexSubImage2D, CopyTexSubImage2D, TexSubImage1D, or CopyTexSubImage1D will result in an INVALID_OPERATION error if <xoffset>, <yoffset>, or <zoffset> is not equal to -b_s (border). In addition, the contents of any texel outside the region modified by such a call are undefined. These restrictions may be relaxed for specific compressed internal formats whose images are easily edited.

(add new subsection at end of section, p.123)

Compressed Texture Images

Texture images may also be specified or modified using image data already stored in a known compressed image format. The ARB_texture_compression extension defines no such formats, but provides the mechanisms for other extensions that do.

The commands

```
  void CompressedTexImage1DARB(enum target, int level,
                               enum internalformat, sizei width,
                               int border, sizei imageSize,
                               const void *data);
  void CompressedTexImage2DARB(enum target, int level,
                               enum internalformat, sizei width,
                               sizei height, int border,
                               sizei imageSize, const void *data);
  void CompressedTexImage3DARB(enum target, int level,
                               enum internalformat, sizei width,
                               sizei height, sizei depth,
                               int border, sizei imageSize,
                               const void *data);
```

define one-, two-, and three-dimensional texture images, respectively, with incoming data stored in a specific compressed image format. The <target>, <level>, <internalformat>, <width>, <height>, <depth>, and <border> parameters have the same meaning as in TexImage1D, TexImage2D, and TexImage3D. <data> points to compressed image data stored in the compressed image format corresponding to <internalformat>. Since this extension provides no specific image formats, using any of the six generic compressed internal formats as <internalformat> will result in an INVALID_ENUM error.

For all other compressed internal formats, the compressed image will be decoded according to the specification defining the <internalformat> token. Compressed texture images are treated as an array of <imageSize> ubytes beginning at address <data>. All pixel storage and pixel transfer modes are ignored when decoding a compressed texture image. If the <imageSize> parameter is not consistent with the format, dimensions, and contents of the compressed image, an INVALID_VALUE error results. If the compressed image is not encoded according to the defined image format, the results of the call are undefined.

Specific compressed internal formats may impose format-specific restrictions on the use of the compressed image specification calls or parameters. For example, the compressed image format might be supported only for 2D textures or may not allow non-zero <border> values. Any such restrictions will be documented in the specification defining the compressed internal format; violating these restrictions will result in an INVALID_OPERATION error.

Any restrictions imposed by specific compressed internal formats will be invariant, meaning that if the GL accepts and stores a texture image in compressed form, providing the same image to CompressedTexImage1DARB, CompressedTexImage2DARB, CompressedTexImage3DARB will not result in an INVALID_OPERATION error if the following restrictions are satisfied:

* <data> points to a compressed texture image returned by GetCompressedTexImageARB (Section 6.1.4).

* <target>, <level>, and <internalformat> match the <target>, <level> and <format> parameters provided to the GetCompressedTexImageARB call returning <data>.

* <width>, <height>, <depth>, <border>, <internalformat>, and
  <imageSize> match the values of TEXTURE_WIDTH, TEXTURE_HEIGHT,
  TEXTURE_DEPTH, TEXTURE_BORDER, TEXTURE_INTERNAL_FORMAT, and
  TEXTURE_COMPRESSED_IMAGE_SIZE_ARB for image level <level> in effect at
  the time of the GetCompressedTexImageARB call returning <data>.

This guarantee applies not just to images returned by
GetCompressedTexImageARB, but also to any other properly encoded
compressed texture image of the same size and format.

The commands

```
  void CompressedTexSubImage1DARB(enum target, int level,
                                  int xoffset, sizei width,
                                  enum format, sizei imageSize,
                                  const void *data);
  void CompressedTexSubImage2DARB(enum target, int level,
                                  int xoffset, int yoffset,
                                  sizei width, sizei height,
                                  enum format, sizei imageSize,
                                  const void *data);
  void CompressedTexSubImage3DARB(enum target, int level,
                                  int xoffset, int yoffset,
                                  int zoffset, sizei width,
                                  sizei height, sizei depth,
                                  enum format, sizei imageSize,
                                  const void *data);
```

respecify only a rectangular region of an existing texture array, with
incoming data stored in a known compressed image format. The <target>,
<level>, <xoffset>, <yoffset>, <zoffset>, <width>, <height>, and <depth>
parameters have the same meaning as in TexSubImage1D, TexSubImage2D, and
TexSubImage3D. <data> points to compressed image data stored in the
compressed image format corresponding to <format>. Since this extension
provides no specific image formats, using any of these six generic
compressed internal formats as <format> will result in an INVALID_ENUM
error.

The image pointed to by <data> and the <imageSize> parameter are
interpreted as though they were provided to CompressedTexImage1DARB,
CompressedTexImage2DARB, and CompressedTexImage3DARB. These commands do
not provide for image format conversion, so an INVALID_OPERATION error
results if <format> does not match the internal format of the texture
image being modified. If the <imageSize> parameter is not consistent with
the format, dimensions, and contents of the compressed image (too little
or too much data), an INVALID_VALUE error results.

As with CompressedTexImage calls, compressed internal formats may have
additional restrictions on the use of the compressed image specification
calls or parameters. Any such restrictions will be documented in the
specification defining the compressed internal format; violating these
restrictions will result in an INVALID_OPERATION error.

Any restrictions imposed by specific compressed internal formats will be
invariant, meaning that if the GL accepts and stores a texture image in

compressed form, providing the same image to CompressedTexSubImage1DARB, CompressedTexSubImage2DARB, CompressedTexSubImage3DARB will not result in an INVALID_OPERATION error if the following restrictions are satisfied:

* <data> points to a compressed texture image returned by GetCompressedTexImageARB (Section 6.1.4).

* <target>, <level>, and <format> match the <target>, <level> and <format> parameters provided to the GetCompressedTexImageARB call returning <data>.

* <width>, <height>, <depth>, <format>, and <imageSize> match the values of TEXTURE_WIDTH, TEXTURE_HEIGHT, TEXTURE_DEPTH, TEXTURE_INTERNAL_FORMAT, and TEXTURE_COMPRESSED_IMAGE_SIZE_ARB for image level <level> in effect at the time of the GetCompressedTexImageARB call returning <data>.

* <width>, <height>, <depth>, <format> match the values of TEXTURE_WIDTH, TEXTURE_HEIGHT, TEXTURE_DEPTH, and TEXTURE_INTERNAL_FORMAT currently in effect for image level <level>.

* <xoffset>, <yoffset>, and <zoffset> are all "-<b>", where <b> is the value of TEXTURE_BORDER currently in effect for image level <level>.

This guarantee applies not just to images returned by GetCompressedTexImageARB, but also to any other properly encoded compressed texture image of the same size.

Calling CompressedTexSubImage3D, CompressedTexSubImage2D, or CompressedTexSubImage1D will result in an INVALID_OPERATION error if <xoffset>, <yoffset>, or <zoffset> is not equal to -b_s (border), or if <width>, <height>, and <depth> do not match the values of TEXTURE_WIDTH, TEXTURE_HEIGHT, or TEXTURE_DEPTH, respectively. The contents of any texel outside the region modified by the call are undefined. These restrictions may be relaxed for specific compressed internal formats whose images are easily edited.

Additions to Chapter 4 of the OpenGL 1.2.1 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 1.2.1 Specification (Special Functions)

Modify **Section 5.6, Hints** (p.180)

(p.180, modify first paragraph)

...; FOG_HINT, indicating whether fog calculations are done per pixel or per vertex; and TEXTURE_COMPRESSION_HINT_ARB, indicating the desired quality and performance of compressing texture images.

For the texture compression hint, a <hint> of FASTEST indicates that texture images should be compressed as quickly as possible, while NICEST indicates that the texture images be compressed with as little image degradation as possible. FASTEST should be used for one-time texture compression, and NICEST should be used if the compression results are to be retrieved by GetCompressedTexImageARB (Section 6.1.4) for reuse.

Additions to Chapter 6 of the OpenGL 1.2.1 Specification (State and State Requests)

Modify **Section 6.1.3, Enumerated Queries** (p.183)

(p.183, modify next-to-last paragraph)

For texture images with uncompressed internal formats, queries of TEXTURE_RED_SIZE, TEXTURE_GREEN_SIZE, TEXTURE_BLUE_SIZE, TEXTURE_ALPHA_SIZE, TEXTURE_LUMINANCE_SIZE, and TEXTURE_INTENSITY_SIZE return the actual resolutions of the stored image array components, not the resolutions specified when the image array was defined. For texture images with a compressed internal format, the resolutions returned specify the component resolution of an uncompressed internal format that produces an image of roughly the same quality as the compressed image in question. Since the quality of the implementation's compression algorithm is likely data-dependent, the returned component sizes should be treated only as rough approximations. ...

(p.183, add to end of next-to-last paragraph)

TEXTURE_COMPRESSED_IMAGE_SIZE_ARB returns the size (in ubytes) of the compressed texture image that would be returned by GetCompressedTexImageARB (Section 6.1.4). Querying TEXTURE_COMPRESSED_IMAGE_SIZE_ARB is not allowed on texture images with an uncompressed internal format or on proxy targets and will result in an INVALID_OPERATION error if attempted.

Modify **Section 6.1.4, Texture Queries** (p.184)

(add immediately after the GetTexImage section and before the IsTexture section)

The command

```
  void GetCompressedTexImageARB(enum target, int lod,
                                void *img);
```

is used to obtain texture images stored in compressed form. The parameters <target>, <lod>, and <img> are interpreted in the same manner as in GetTexImage. When called, GetCompressedTexImageARB writes TEXTURE_COMPRESSED_IMAGE_SIZE_ARB ubytes of compressed image data to the memory pointed to by <img>. The compressed image data is formatted according to the specification defining INTERNAL_FORMAT. All pixel storage and pixel transfer modes are ignored when returning a compressed texture image.

Calling GetCompressedTexImageARB with an <lod> value less than zero or greater than the maximum allowable causes an INVALID_VALUE error. Calling GetCompressedTexImageARB with a texture image stored with an uncompressed internal format causes an INVALID_OPERATION error.

Additions to Appendix A of the OpenGL 1.2.1 Specification (Invariance)

None.

Additions to the AGL/GLX/WGL Specifications

None.

GLX Protocol

(Add after GetTexImage to Section 2.2.2 of the GLX 1.3 encoding spec, p.74)

GetCompressedTexImageARB

```
    1       CARD8               opcode (X assigned)
    1       160                 GLX opcode
    2       4                   request length
    4       GLX_CONTEXT_TAG     context tag
    4       ENUM                target
    4       INT32               level

 -->

    1       1                   Reply
    1       1                   unused
    2       CARD16              sequence number
    4       n                   reply length
    8                           unused
    4       INT32               compressed image size (in bytes) --
                                  should be between 4n-3 and 4n
    12                          unused
    4*n     LISTofBYTE          teximage
```

Note that n may be zero, indicating that a GL error occurred.

Since pixel storage modes do not apply to compressed texture images, teximage is simply an array of bytes. The client library will ignore pixel storage modes and should copy only <compressed image size> bytes, regardless of the value of <reply length>.

```
(Add to end of Section 2.3 of the GLX 1.3 encoding spec, p.147)
```

CompressedTexImage1DARB

```
    2       32+n+p          rendering command length
    2       214             rendering command opcode
    4       ENUM            target
    4       INT32           level
    4       ENUM            internalformat
    4       INT32           width
    4                       unused
    4       INT32           border
    n       LISTofBYTE      image
    4       INT32           imageSize
    p                       unused, p=pad(n)
```

```
If the command is encoded in a glXRenderLarge request, the command
opcode and command length fields are expanded to 4 bytes each.
```

```
    4       36+n+p          rendering command length
    4       214             rendering command opcode
```

CompressedTexImage2DARB

```
    2       32+n+p          rendering command length
    2       215             rendering command opcode
    4       ENUM            target
    4       INT32           level
    4       ENUM            internalformat
    4       INT32           width
    4       INT32           height
    4       INT32           border
    4       INT32           imageSize
    n       LISTofBYTE      image
    p                       unused, p=pad(n)
```

```
If the command is encoded in a glXRenderLarge request, the command
opcode and command length fields are expanded to 4 bytes each.
```

```
    4       36+n+p          rendering command length
    4       215             rendering command opcode
```

CompressedTexImage3DARB

```
    2       36+n+p          rendering command length
    2       216             rendering command opcode
    4       ENUM            target
    4       INT32           level
    4       INT32           internalformat
    4       INT32           width
    4       INT32           height
    4       INT32           depth
    4       INT32           border
    4       INT32           imageSize
    n       LISTofBYTE      image
    p                       unused, p=pad(n)
```

If the command is encoded in a glXRenderLarge request, the command opcode and command length fields are expanded to 4 bytes each.

```
    4       36+n+p          rendering command length
    4       216             rendering command opcode
```

CompressedTexSubImage1DARB

```
    2       36+n+p          rendering command length
    2       217             rendering command opcode
    4       ENUM            target
    4       INT32           level
    4       INT32           xoffset
    4                       unused
    4       INT32           width
    4                       unused
    4       ENUM            format
    4       INT32           imageSize
    n       LISTofBYTE      image
    p                       unused, p=pad(n)
```

If the command is encoded in a glXRenderLarge request, the command opcode and command length fields are expanded to 4 bytes each.

```
    4       40+n+p          rendering command length
    4       217             rendering command opcode
    CompressedTexSubImage2DARB

        2       36+n+p          rendering command length
        2       218             rendering command opcode
        4       ENUM            target
        4       INT32           level
        4       INT32           xoffset
        4       INT32           yoffset
        4       INT32           width
        4       INT32           height
        4       ENUM            format
        4       INT32           imageSize
        n       LISTofBYTE      image
        p                       unused, p=pad(n)

    If the command is encoded in a glXRenderLarge request, the command
    opcode and command length fields are expanded to 4 bytes each.

        4       40+n+p          rendering command length
        4       218             rendering command opcode

    CompressedTexSubImage3DARB

        2       44+n+p          rendering command length
        2       219             rendering command opcode
        4       ENUM            target
        4       INT32           level
        4       INT32           xoffset
        4       INT32           yoffset
        4       INT32           zoffset
        4       INT32           width
        4       INT32           height
        4       INT32           depth
        4       ENUM            format
        4       INT32           imageSize
        n       LISTofBYTE      image
        p                       unused, p=pad(n)

    If the command is encoded in a glXRenderLarge request, the command
    opcode and command length fields are expanded to 4 bytes each.

        4       48+n+p          rendering command length
        4       219             rendering command opcode
```

Errors

```
    Errors for compressed TexImage and TexSubImage calls specific to
    compression:

    INVALID_OPERATION is generated by TexSubImage1D, TexSubImage2D,
    TexSubImage3D, CopyTexSubImage1D, CopyTexSubImage2D, or CopyTexSubImage3D
    if the internal format of the texture image is compressed and <xoffset>,
    <yoffset>, or <zoffset> does not equal -b, where b is value of
    TEXTURE_BORDER.
    INVALID_VALUE is generated by CompressedTexSubImage1DARB,
    CompressedTexSubImage2DARB, or CompressedTexSubImage3DARB if the entire
    texture image is not being edited:  if <xoffset>, <yoffset>, or <zoffset>
    is greater than -b, <xoffset> + <width> is less than w+b, <yoffset> +
    <height> is less than h+b, or <zoffset> + <depth> is less than d+b, where
    b is the value of TEXTURE_BORDER, w is the value of TEXTURE_WIDTH, h is
    the value of TEXTURE_HEIGHT, and d is the value of TEXTURE_DEPTH.

    INVALID_ENUM is generated by CompressedTexImage1DARB,
    CompressedTexImage2DARB, or CompressedTexImage3DARB,
    CompressedTexSubImage1DARB, CompressedTexSubImage2DARB, or
    CompressedTexSubImage3DARB, if <internalformat> is any of the six generic
    compressed internal formats (e.g., COMPRESSED_RGBA_ARB)

    INVALID_OPERATION is generated by CompressedTexImage1DARB,
    CompressedTexImage2DARB, CompressedTexImage3DARB,
    CompressedTexSubImage1DARB, CompressedTexSubImage2DARB, or
    CompressedTexSubImage3DARB, if any parameter combinations are not
    supported by the specific compressed internal format.  Such invalid
    combinations are documented in the specification defining the internal
    format.

    INVALID_VALUE is generated by CompressedTexImage1DARB,
    CompressedTexImage2DARB, or CompressedTexImage3DARB,
    CompressedTexSubImage1DARB, CompressedTexSubImage2DARB, or
    CompressedTexSubImage3DARB, if <imageSize> is not consistent with the
    format, dimensions, and contents of the specified image.  The appropriate
    value for the <imageSize> parameter is documented in the specification
    defining the compressed internal format.

    Undefined results (including abnormal program termination) are generated
    by CompressedTexImage1DARB, CompressedTexImage2DARB, or
    CompressedTexImage3DARB, CompressedTexSubImage1DARB,
    CompressedTexSubImage2DARB, or CompressedTexSubImage3DARB, is not encoded
    in a manner consistent with the specification defining the internal
    format.

    INVALID_OPERATION is generated by CompressedTexSubImage1DARB,
    CompressedTexSubImage2DARB, or CompressedTexSubImage3DARB if <format> does
    not match the internal format of the texture image being modified.

    INVALID_OPERATION is generated by GetTexLevelParameter[if]v if <target> is
    PROXY_TEXTURE_1D, PROXY_TEXTURE_2D, or PROXY_TEXTURE_3D and <value> is
    TEXTURE_COMPRESSED_IMAGE_SIZE_ARB.

    INVALID_OPERATION is generated by GetTexLevelParameter[if]v if the
    internal format of the queried texture image is not compressed and <value>
    is TEXTURE_COMPRESSED_IMAGE_SIZE_ARB.

    INVALID_OPERATION is generated by GetCompressedTexImageARB if the internal
    format of the queried texture image is not compressed.


    Errors for compressed TexImage and TexSubImage calls not specific to
    compression:
```

```
    INVALID_ENUM is generated by CompressedTexImage3DARB or
    CompressedTexSubImage3DARB if <target> is not TEXTURE_3D.

    INVALID_ENUM is generated by CompressedTexImage2DARB or
    CompressedTexSubImage2DARB if <target> is not TEXTURE_2D,
    TEXTURE_CUBE_MAP_POSITIVE_X_ARB, TEXTURE_CUBE_MAP_NEGATIVE_X_ARB,
    TEXTURE_CUBE_MAP_POSITIVE_Y_ARB, TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB,
    TEXTURE_CUBE_MAP_POSITIVE_Z_ARB, or TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB.

    INVALID_ENUM is generated by CompressedTexImage1DARB or
    CompressedTexSubImage1DARB if <target> is not TEXTURE_1D.

    INVALID_VALUE is generated by CompressedTexImage1DARB,
    CompressedTexImage2DARB, CompressedTexImage3DARB,
    CompressedTexSubImage1DARB, CompressedTexSubImage1DARB, or
    CompressedTexSubImage3DARB if <level> is negative.

    INVALID_VALUE is generated by CompressedTexImage1DARB,
    CompressedTexImage2DARB, CompressedTexImage3DARB,
    CompressedTexSubImage1DARB, CompressedTexSubImage2DARB, or
    CompressedTexSubImage3DARB, if <width>, <height>, or <depth> is negative.

    INVALID_VALUE is generated by CompressedTexImage1DARB,
    CompressedTexImage2DARB, or CompressedTexImage3DARB if <width>, <height>,
    or <depth> can not be represented as 2^k+2 for some integer value k.

    INVALID_VALUE is generated by CompressedTexImage1DARB,
    CompressedTexImage2DARB, or CompressedTexImage3DARB if <border> is not
    zero or one.

    INVALID_VALUE is generated by CompressedTexImage1DARB,
    CompressedTexImage2DARB, CompressedTexImage3DARB,
    CompressedTexSubImage1DARB, CompressedTexSubImage1DARB, or
    CompressedTexSubImage3DARB if the call is made between a call to Begin and
    the corresponding call to End.

    INVALID_VALUE is generated by CompressedTexSubImage1DARB,
    CompressedTexSubImage2DARB, or CompressedTexSubImage3DARB if <xoffset>,
    <yoffset>, or <zoffset> is less than -b, <xoffset> + <width> is greater
    than w+b, <yoffset> + <height> is greater than h+b, or <zoffset> + <depth>
    is greater than d+b, where b is the value of TEXTURE_BORDER, w is the
    value of TEXTURE_WIDTH, h is the value of TEXTURE_HEIGHT, and d is the
    value of TEXTURE_DEPTH.

    INVALID_VALUE is generated by GetCompressedTexImageARB if <lod> is
    negative or greater than the maximum allowable level.
```

New State

(table 6.12, p.202)

Get Value	Type	Get Command	Initial Value	Description	Sec.	Attribute
TEXTURE_COMPRESSED_IMAGE_SIZE_ARB	n x Z+	GetTexLevel-Parameter	0	size (in ubytes) of xD compressed texture image i.	3.8	-
TEXTURE_COMPRESSED_ARB	n x B	GetTexLevel-Parameter	FALSE	True if xD image i has a compressed internal format	3.8	-

(table 6.23, p.213)

Get Value	Type	Get Command	Initial Value	Description	Sec.	Attribute
TEXTURE_COMPRESSION_HINT_ARB	Z_3	GetIntegerv	DONT_ CARE	Texture compression quality hint	5.6	hint

(table 6.25, p. 215)

Get Value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
NUM_COMPRESSED_TEXTURE_FORMATS_ARB	Z	GetIntegerv	0	Number of enumerated compressed texture formats	3.8	-
COMPRESSED_TEXTURE_FORMATS_ARB	0* x Z	GetIntegerv	-	Enumerated compressed texture formats	3.8	-

Revision History

```
    1.03, 05/23/00 prbrown1: Removed stray "None." paragraph in modifications
                             to Chapter 5.

    1.02, 05/08/00 prbrown1: Fixed prototype of GetCompressedTexImageARB (no
                             "const" qualifiers) in "New Procedures and
                             Functions" section.  Changed <internalformat>
                             parameter of CompressedTexImage functions to be
                             an "enum" instead of an "int".  "int" was carried
                             over only on TexImage calls as a 1.0 legacy --
                             the newer CopyTexImage call takes an "enum".

    1.01, 04/11/00 prbrown1: Minor bug fixes to the first published version.
                             Fixed prototypes to match extension spec
                             standards (no "GL" type prefixes).  Fixed a
                             couple erroneous function names.  Added "const"
                             qualifier to prototypes involving image data not
                             modified by the GL.  Added text to indicate that
                             compressed formats apply to texture maps
                             supported by GL_ARB_texture_cube_map.
    1.0,  03/24/00 prbrown1: Applied changes approved as part of the extension
                             at the March 2000 ARB meeting, as follows:

                             * CompressedTexSubImage:  Only allowed if the
                               entire image is replaced.  Document that this
                               restriction can be relaxed for specific
                               compression extensions.
                             * Renamed TEXTURE_IMAGE_SIZE_ARB to
                               TEXTURE_COMPRESSED_IMAGE_SIZE_ARB.
                             * Querying image size on uncompressed images is
                               now an INVALID_OPERATION error.
                             * INVALID_VALUE error is generated if <imageSize>
                               is inconsistent with the image data.  This
                               restriction may be overridden by specific
                               extensions only if requiring an image size
                               check is unreasonable.
                             * Added documentaion of undefined behavior for
                               CompressedTexImage/SubImage if the image data
                               is encoded in a manner inconsistent with the
                               spec defining the compressed image format.
                             * Fixed issue (16).  Text was truncated.
                             * Modified invariance section.  <data> can not
                               affect the choice of compressed internal
                               format, but can theoretically affect regular
                               component resolution.
                             * Add new function GetCompressedTexImage to deal
                               with subtle GLX issues.
                             * GLX protocol for CompressedTexImage/SubImage
                               and GetCompressedTexImage holds both a padded
                               image size (for GLX data transfer) and actual
                               image size (for packing in user buffers).

                             Minor wording clean-ups.

                             Added enum and GLX opcode values allocated from
                             OpenGL Extensions and GLX registries.

    0.81, 03/07/00 prbrown1: Fixed error documentation for TexSubImage calls
                             of arbitrary alignment (did not document that the
                             internal format had to be compressed).  Removed
                             references to CopyTexImage3D, which doesn't
                             actually exist.

                             Per Kurt Akeley suggestions: (1) Renamed
                             TexImageCompressed to CompressedTexImage to
                             conform with naming conventions, (2) clarified
                             that the main feature distinguishing
                             CompressedTex[Sub]Image calls from normal
                             Tex[Sub]Image calls is compressed input data, (3)
                             added query to explicitly determine whether the
                             internal format of a texture is compressed.

    0.8,  02/23/00 prbrown1: Marked previously unresolved issues as resolved
                             per the ARB working group.  Added docs for errors
                             not specific to compression for the new
                             CompressedTexImage and CompressedTexSubImage
                             calls.  Added queries to enumerate specific
```

```
                                 compressed texture formats.
    0.76, 02/16/00 prbrown1: Removed "gl" and "GL_" prefixes.
    0.75, 02/07/00 prbrown1: Incorporated feedback from 12/99 ARB meeting
                             and a number of other revisions.
    0.7,  12/03/99 prbrown1: Incorporated comments from public review of 0.2
                             document.
    0.2,  10/28/99 prbrown1: Renamed to ARB_texture_compression.  Significant
                             functional changes.
    0.11, 10/21/99 prbrown1: Edits suggested by 3dfx.
    0.1,  10/19/99 prbrown1: Initial revision.
```

Name

    ARB_texture_cube_map

Name Strings

    GL_ARB_texture_cube_map

Notice

Status

    Complete. Approved by ARB on 12/8/1999

Version

    Last Modified Date: December 14, 1999

Number

    ARB Extension #7

Dependencies

    None.

    Written based on the wording of the OpenGL 1.2.1 specification but
    not dependent on it.

Overview

This extension provides a new texture generation scheme for cube map textures. Instead of the current texture providing a 1D, 2D, or 3D lookup into a 1D, 2D, or 3D texture image, the texture is a set of six 2D images representing the faces of a cube. The (s,t,r) texture coordinates are treated as a direction vector emanating from the center of a cube. At texture generation time, the interpolated per-fragment (s,t,r) selects one cube face 2D image based on the largest magnitude coordinate (the major axis). A new 2D (s,t) is calculated by dividing the two other coordinates (the minor axes values) by the major axis value. Then the new (s,t) is used to lookup into the selected 2D texture image face of the cube map.

Unlike a standard 1D, 2D, or 3D texture that have just one target, a cube map texture has six targets, one for each of its six 2D texture image cube faces. All these targets must be consistent, complete, and have equal width and height (ie, square dimensions).

This extension also provides two new texture coordinate generation modes for use in conjunction with cube map texturing. The reflection map mode generates texture coordinates (s,t,r) matching the vertex's eye-space reflection vector. The reflection map mode is useful for environment mapping without the singularity inherent in sphere mapping. The normal map mode generates texture coordinates (s,t,r) matching the vertex's transformed eye-space normal. The normal map mode is useful for sophisticated cube map texturing-based diffuse lighting models.

The intent of the new texgen functionality is that an application using cube map texturing can use the new texgen modes to automatically generate the reflection or normal vectors used to look up into the cube map texture.

An application note: When using cube mapping with dynamic cube maps (meaning the cube map texture is re-rendered every frame), by keeping the cube map's orientation pointing at the eye position, the texgen-computed reflection or normal vector texture coordinates can be always properly oriented for the cube map. However if the cube map is static (meaning that when view changes, the cube map texture is not updated), the texture matrix must be used to rotate the texgen-computed reflection or normal vector texture coordinates to match the orientation of the cube map. The rotation can be computed based on two vectors: 1) the direction vector from the cube map center to the eye position (both in world coordinates), and 2) the cube map orientation in world coordinates. The axis of rotation is the cross product of these two vectors; the angle of rotation is the arcsin of the dot product of these two vectors.

Issues

Should we place the normal/reflection vector in the (s,t,r) texture coordinates or (s,t,q) coordinates?

RESOLUTION: (s,t,r). Even if hardware uses "q" for the third component, the API should claim to support generation of (s,t,r) and let the texture matrix (through a concatenation with the user-supplied texture matrix) move "r" into "q".

Should the texture coordinate generation functionality for cube mapping be specified as a distinct extension from the actual cube map texturing functionality?

RESOLUTION: NO. Real applications and real implementations of cube mapping will tie the texgen and texture generation functionality together. Applications won't have to query two separate extensions then.

While applications will almost always want to use the texgen functionality for automatically generating the reflection or normal vector as texture coordinates (s,t,r), this extension does permit an application to manually supply the reflection or normal vector through glTexCoord3f explicitly.

Note that the NV_texgen_reflection extension does "unbundle" the texgen functionality from cube maps.

Should you be able to have some texture coordinates computing REFLECTION_MAP_ARB and others not? Same question with NORMAL_MAP_ARB.

RESOLUTION: YES. This is the way that SPHERE_MAP works. It is not clear that this would ever be useful though.

Should something special be said about the handling of the q texture coordinate for this spec?

RESOLUTION: NO. But the following paragraph is useful for implementors concerned about the handling of q.

The REFLECTION_MAP_ARB and NORMAL_MAP_ARB modes are intended to supply reflection and normal vectors for cube map texturing hardware. When these modes are used for cube map texturing, the generated texture coordinates can be thought of as an reflection vector. The value of the q texture coordinate then simply scales the vector but does not change its direction. Because only the vector direction (not the vector magnitude) matters for cube map texturing, implementations are free to leave q undefined when any of the s, t, or r texture coordinates are generated using REFLECTION_MAP_ARB or NORMAL_MAP_ARB.

How should the cube faces be labeled?

RESOLUTION: Match the render man specification's names of "px" (positive X), "nx" (negative x), "py", "ny", "pz", and "nz". There does not actually need to be an "ordering for the faces" (Direct3D 7.0 does number their cube map faces.) For this extension, the symbolic target names (TEXTURE_CUBE_MAP_POSITIVE_X_ARB, etc) is sufficient without requiring any specific ordering.

What coordinate system convention should be used? LHS or RHS?

RESOLUTION: The coordinate system is left-handed if you think of yourself within the cube. The coordinate system is right-handed if you think of yourself outside the cube.

This matches the convention of the RenderMan interface. If you look at Figure 12.8 (page 265) in "The RenderMan Companion", think of the cube being folded up with the observer inside the cube. Then the coordinate system convention is left-handed.

The spec just linearly interpolates the reflection vectors computed per-vertex across polygons. Is there a problem interpolating reflection vectors in this way?

Probably. The better approach would be to interpolate the eye vector and normal vector over the polygon and perform the reflection vector computation on a per-fragment basis. Not doing so is likely to lead to artifacts because angular changes in the normal vector result in twice as large a change in the reflection vector as normal vector changes. The effect is likely to be reflections that become glancing reflections too fast over the surface of the polygon.

Note that this is an issue for REFLECTION_MAP_ARB, but not NORMAL_MAP_ARB.

What happens if an (s,t,q) is passed to cube map generation that is close to (0,0,0), ie. a degenerate direction vector?

RESOLUTION: Leave undefined what happens in this case (but may not lead to GL interruption or termination).

Note that a vector close to (0,0,0) may be generated as a result of the per-fragment interpolation of (s,t,r) between vertices.

Do we need a distinct proxy texture mechanism for cube map textures?

RESOLUTION: YES. Cube map textures take up six times the memory as a conventional 2D image texture so proxy 2D texture determinations won't be of value for a cube map texture. Cube maps need their own proxy target.

Should we require the 2D texture image width and height to be identical (ie, square only)?

RESOLUTION: YES. This limitation is quite a reasonable limitation and DirectX 7 has the same limitation.

This restriction is enforced by generating an INVALID_VALUE when calling TexImage2D or CopyTexImage2D with a non-equal width and height.

Some consideration was given to enforcing the "squarness" constraint as a texture consistency constraint. This is confusing however since the squareness is known up-front at texture image specification time so it seems confusing to silently report the usage error as a texture consistency issue.

Texture consistency still says that all the level 0 textures of all six faces must have the same square size.

If some combination of 1D, 2D, 3D, and cube map texturing is enabled, which really operates?

RESOLUTION: Cube map texturing. In OpenGL 1.2, 3D takes priority over 2D takes priority over 1D. Cube mapping should take priority over all conventional n-dimensional texturing schemes.

Does anything need to be said about combining cube mapping with multitexture?

RESOLUTION: NO. Cube mapping should be available on all texture units. The hardware should fully orthogonal in its handling of cube map textures.

Does it make sense to support borders for cube map textures.

Actually, it does. It would be nice if the texture border pixels match the appropriate texels from the edges of the other cube map faces that they junction with. For this reason, we'll leave the texture border capability implicitly supported.

How does mipmap level-of-detail selection work for cube map textures?

The existing spec's language about LOD selection is fine.

Should the implementation dependent value for the maximum texture size for a cube map be the same as MAX_TEXTURE_SIZE?

RESOLUTION: NO. OpenGL 1.2 has a different MAX_3D_TEXTURE_SIZE for 3D textures, and cube maps should take six times more space than a 2D texture map of the same width & height. The implementation dependent MAX_CUBE_MAP_TEXTURE_SIZE_ARB constant should be used for cube maps then.

Note that the proxy cube map texture provides a better way to find out the maximum cube map texture size supported since the proxy mechanism can take into account the internal format, etc.

In section 3.8.10 when the "largest magnitude coordinate direction" is choosen, what happens if two or more of the coordinates (rx,ry,rz) have the identical magnitude?

RESOLUTION: Implementations can define their own rule to choose the largest magnitude coordinate direction whne two or more of the coordinates have the identical magnitude. The only restriction is that the rule must be deterministic and depend only on (rx,ry,rz).

In practice, (s,t,r) is interpolated across polygons so the cases where |s|==|t|, etc. are pretty arbitary (the equality depends on interpolation precision). This extension could mandate a particular rule, but that seems heavy-handed and there is no good reason that multiple vendors should be forced to implement the same rule.

Should there be limits on the supported border modes for cube maps?

RESOLUTION: NO. The specificiation is written so that cube map texturing proceeds just like conventional 2D texture mapping once the face determination is made.

Therefore, all OpenGL texture wrap modes should be supported though some modes are clearly inappropriate for cube maps. The WRAP mode is almost certainly incorrect for cube maps. Likewise, the CLAMP mode without a texture border is almost certainly incorrect for cube maps. CLAMP when a texture border is present and CLAMP_TO_EDGE are both reasonably suited for cube maps. Ideally, CLAMP with a texture border works best if the cube map edges can be replicated in the approriate texture borders of adjacent cube map faces. In practice, CLAMP_TO_EDGE works reasonably well in most circumstances.

Perhaps another extension could support a special cube map wrap mode that automatically wraps individual texel fetches to the appropriate adjacent cube map face. The benefit from such a mode is small and the implementation complexity is involved so this wrap mode should not be required for a basic cube map texture extension.

How is mipmap LOD selection handled for cube map textures?

RESOLUTION: The specification is written so that cube map texturing proceeds just like conventional 2D texture mapping once the face determination is made.

Thereforce, the partial differentials in Section 3.8.5 (page 126) should be evaluated for the u and v parameters based on the post-face determination s and t.

In Section 2.10.3 "Normal Transformation", there are several versions of the eye-space normal vector to choose from. Which one should the NORMAL_MAP_ARB texgen mode use?

RESOLUTION: nf. The nf vector is the final normal, post-rescale normal and post-normalize. In practice, the rescale normal and normalize operations do not change the direction of the vector so the choice of which version of transformed normal is used is not important for cube maps.

New Procedures and Functions

None

New Tokens

Accepted by the <param> parameters of TexGend, TexGenf, and TexGeni when <pname> parameter is TEXTURE_GEN_MODE:

```
NORMAL_MAP_ARB                     0x8511
REFLECTION_MAP_ARB                 0x8512
```

When the <pname> parameter of TexGendv, TexGenfv, and TexGeniv is TEXTURE_GEN_MODE, then the array <params> may also contain NORMAL_MAP_ARB or REFLECTION_MAP_ARB.

Accepted by the <cap> parameter of Enable, Disable, IsEnabled, and by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev, and by the <target> parameter of BindTexture, GetTexParameterfv, GetTexParameteriv, TexParameterf, TexParameteri, TexParameterfv, and TexParameteriv:

```
TEXTURE_CUBE_MAP_ARB               0x8513
```

Accepted by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

```
TEXTURE_BINDING_CUBE_MAP_ARB       0x8514
```

```
    Accepted by the <target> parameter of GetTexImage,
    GetTexLevelParameteriv, GetTexLevelParameterfv, TexImage2D,
    CopyTexImage2D, TexSubImage2D, and CopySubTexImage2D:

        TEXTURE_CUBE_MAP_POSITIVE_X_ARB     0x8515
        TEXTURE_CUBE_MAP_NEGATIVE_X_ARB     0x8516
        TEXTURE_CUBE_MAP_POSITIVE_Y_ARB     0x8517
        TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB     0x8518
        TEXTURE_CUBE_MAP_POSITIVE_Z_ARB     0x8519
        TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB     0x851A

    Accepted by the <target> parameter of GetTexLevelParameteriv,
    GetTexLevelParameterfv, GetTexParameteriv, and TexImage2D:

        PROXY_TEXTURE_CUBE_MAP_ARB          0x851B

    Accepted by the <pname> parameter of GetBooleanv, GetDoublev,
    GetIntegerv, and GetFloatv:

        MAX_CUBE_MAP_TEXTURE_SIZE_ARB       0x851C
```

Additions to Chapter 2 of the 1.2 Specification (OpenGL Operation)

-- Section 2.10.4 "Generating Texture Coordinates"

```
      Change the last sentence in the 1st paragraph (page 37) to:

      "If <pname> is TEXTURE_GEN_MODE, then either <params> points to
      or <param> is an integer that is one of the symbolic constants
      OBJECT_LINEAR, EYE_LINEAR, SPHERE_MAP, REFLECTION_MAP_ARB, or
      NORMAL_MAP_ARB."

      Add these paragraphs after the 4th paragraph (page 38):

      "If TEXTURE_GEN_MODE indicates REFLECTION_MAP_ARB, compute the
      reflection vector r as described for the SPHERE_MAP mode.  Then the
      value assigned to an s coordinate (the first TexGen argument value
      is S) is s = rx; the value assigned to a t coordinate is t = ry;
      and the value assigned to a r coordinate is r = rz.  Calling TexGen
      with a <coord> of Q when <pname> indicates REFLECTION_MAP_ARB
      generates the error INVALID_ENUM.

      If TEXTURE_GEN_MODE indicates NORMAL_MAP_ARB, compute the normal
      vector nf as described in section 2.10.3.  Then the value assigned
      to an s coordinate (the first TexGen argument value is S) is s =
      nfx; the value assigned to a t coordinate is t = nfy; and the
      value assigned to a r coordinate is r = nfz.  (The values nfx, nfy,
      and nfz are the components of nf.)  Calling TexGen with a <coord>
      of Q when <pname> indicates NORMAL_MAP_ARB generates the error
      INVALID_ENUM.

      The last paragraph's first sentence (page 38) should be changed to:

      "The state required for texture coordinate generation comprises a
      five-valued integer for each coordinate indicating coordinate
      generation mode, ..."
```

Additions to Chapter 3 of the 1.2 Specification (Rasterization)

-- Section 3.6.5 "Pixel Transfer Operations" under "Convolution"

Change this paragraph (page 103) to say:

... "If CONVOLUTION_2D is enabled, the two-dimensional convolution filter is applied only to the two-dimensional images passed to DrawPixels, CopyPixels, ReadPixels, TexImage2D, TexSubImage2D, CopyTexImage2D, CopyTexSubImage2D, and CopyTexSubImage3D, and returned by GetTexImage with one of the targets TEXTURE_2D, TEXTURE_CUBE_MAP_POSITIVE_X_ARB, TEXTURE_CUBE_MAP_NEGATIVE_X_ARB, TEXTURE_CUBE_MAP_POSITIVE_Y_ARB, TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB, TEXTURE_CUBE_MAP_POSITIVE_Z_ARB, or TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB."

-- Section 3.8.1 "Texture Image Specification"

Change the second and third to last sentences on page 116 to:

"<target> must be one of TEXTURE_2D for a 2D texture, or one of TEXTURE_CUBE_MAP_POSITIVE_X_ARB, TEXTURE_CUBE_MAP_NEGATIVE_X_ARB, TEXTURE_CUBE_MAP_POSITIVE_Y_ARB, TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB, TEXTURE_CUBE_MAP_POSITIVE_Z_ARB, or TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB for a cube map texture. Additionally, <target> can be either PROXY_TEXTURE_2D for a 2D proxy texture or PROXY_TEXTURE_CUBE_MAP_ARB for a cube map proxy texture as discussed in section 3.8.7."

Add the following paragraphs after the first paragraph on page 117:

"A 2D texture consists of a single 2D texture image. A cube map texture is a set of six 2D texture images. The six cube map texture targets form a single cube map texture though each target names a distinct face of the cube map. The TEXTURE_CUBE_MAP_*_ARB targets listed above update their appropriate cube map face 2D texture image. Note that the six cube map 2D image tokens such as TEXTURE_CUBE_MAP_POSITIVE_X_ARB are used when specifying, updating, or querying one of a cube map's six 2D image, but when enabling cube map texturing or binding to a cube map texture object (that is when the cube map is accessed as a whole as opposed to a particular 2D image), the TEXTURE_CUBE_MAP_ARB target is specified.

When the target parameter to TexImage2D is one of the six cube map 2D image targets, the error INVALID_VALUE is generated if the width and height parameters are not equal.

If cube map texturing is enabled at the time a primitive is rasterized and if the set of six targets are not "cube complete", then it is as if texture mapping were disabled. The targets of a cube map texture are "cube complete" if the array 0 of all six targets have identical, positive, and square dimensions, the array 0 of all six targets were specified with the same internalformat, and the array 0 of all six targets have the same border width."

After the 14th paragraph (page 116) add:

"In a similiar fashion, the maximum allowable width and height (they must be the same) of a cube map texture must be at least

2^(k-lod)+2bt for image arrays level 0 through k, where k is the
log base 2 of MAX_CUBE_MAP_TEXTURE_SIZE_ARB."

-- Section 3.8.2 "Alternate Texture Image Specification Commands"

Update the second paragraph (page 120) to say:

... "Currently, <target> must be
TEXTURE_2D, TEXTURE_CUBE_MAP_POSITIVE_X_ARB,
TEXTURE_CUBE_MAP_NEGATIVE_X_ARB, TEXTURE_CUBE_MAP_POSITIVE_Y_ARB,
TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB, TEXTURE_CUBE_MAP_POSITIVE_Z_ARB,
or TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB." ...

Add after the second paragraph (page 120), the following:

"When the target parameter to CopyTexImage2D is one of the six cube
map 2D image targets, the error INVALID_VALUE is generated if the
width and height parameters are not equal."

Update the fourth paragraph (page 121) to say:

... "Currently the target arguments of TexSubImage1D and
CopyTexSubImage1D must be TEXTURE_1D, the <target> arguments of
TexSubImage2D and CopyTexSubImage2D must be one of TEXTURE_2D,
TEXTURE_CUBE_MAP_POSITIVE_X_ARB, TEXTURE_CUBE_MAP_NEGATIVE_X_ARB,
TEXTURE_CUBE_MAP_POSITIVE_Y_ARB, TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB,
TEXTURE_CUBE_MAP_POSITIVE_Z_ARB, or TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB,
and the <target> arguments of TexSubImage3D and CopyTexSubImage3D
must be TEXTURE_3D." ...

-- Section 3.8.3 "Texture Parameters"

Change paragraph one (page 124) to say:

... "<target> is the target, either TEXTURE_1D,
TEXTURE_2D, TEXTURE_3D, or TEXTURE_CUBE_MAP_ARB." ...

Add a final paragraph saying:

"Texture parameters for a cube map texture apply to cube map
as a whole; the six distinct 2D texture images use the
texture parameters of the cube map itself.

-- Section 3.8.5 "Texture Minification" under "Mipmapping"

Change the first full paragraph on page 130 to:

... "If texturing is enabled for one-, two-, or three-dimensional
texturing but not cube map texturing (and TEXTURE_MIN_FILTER
is one that requires a mipmap) at the time a primitive is
rasterized and if the set of arrays TEXTURE_BASE_LEVEL through q =
min{p,TEXTURE_MAX_LEVEL} is incomplete, based on the dimensions of
array 0, then it is as if texture mapping were disabled."

```
    Follow the first full paragraph on page 130 with:

    "If cube map texturing is enabled and TEXTURE_MIN_FILTER is one that
    requires mipmap levels at the time a primitive is rasterized and
    if the set of six targets are not "mipmap cube complete", then it
    is as if texture mapping were disabled.  The targets of a cube map
    texture are "mipmap cube complete" if the six cube map targets are
    "cube complete" and the set of arrays TEXTURE_BASE_LEVEL through
    q are not incomplete (as described above)."
```

-- Section 3.8.7 "Texture State and Proxy State"

```
    Change the first sentence of the first paragraph (page 131) to say:

    "The state necessary for texture can be divided into two categories.
    First, there are the nine sets of mipmap arrays (one each for the
    one-, two-, and three-dimensional texture targets and six for the
    cube map texture targets) and their number." ...

    Change the second paragraph (page 132) to say:

    "In addition to the one-, two-, three-dimensional, and the six cube
    map sets of image arrays, the partially instantiated one-, two-,
    and three-dimensional and one cube map sets of proxy image arrays
    are maintained." ...

    After the third paragraph (page 132) add:

    "The cube map proxy arrays are operated on in the same manner
    when TexImage2D is executed with the <target> field specified as
    PROXY_TEXTURE_CUBE_MAP_ARB with the addition that determining that a
    given cube map texture is supported with PROXY_TEXTURE_CUBE_MAP_ARB
    indicates that all six of the cube map 2D images are supported.
    Likewise, if the specified PROXY_TEXTURE_CUBE_MAP_ARB is not
    supported, none of the six cube map 2D images are supported."

    Change the second sentence of the fourth paragraph (page 132) to:

    "Therefore PROXY_TEXTURE_1D, PROXY_TEXTURE_2D, PROXY_TEXTURE_3D,
    and PROXY_TEXTURE_CUBE_MAP_ARB cannot be used as textures, and their
    images must never be queried using GetTexImage." ...
```

-- Section 3.8.8 "Texture Objects"

```
    Change the first sentence of the first paragraph (page 132) to say:

    "In addition to the default textures TEXTURE_1D, TEXTURE_2D,
    TEXTURE_3D, and TEXTURE_CUBE_MAP_ARB, named one-, two-,
    and three-dimensional texture objects and cube map texture objects
    can be created and operated on." ...

    Change the second paragraph (page 132) to say:

    "A texture object is created by binding an unused name to
    TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, or TEXTURE_CUBE_MAP_ARB." ...
    "If the new texture object is bound to TEXTURE_1D, TEXTURE_2D,
```

TEXTURE_3D, or TEXTURE_CUBE_MAP_ARB, it remains a one-, two-, three-dimensional, or cube map texture until it is deleted."

Change the third paragraph (page 133) to say:

"BindTexture may also be used to bind an existing texture object to either TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, or TEXTURE_CUBE_MAP_ARB."

Change paragraph five (page 133) to say:

"In the initial state, TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, and TEXTURE_CUBE_MAP have one-dimensional, two-dimensional, three-dimensional, and cube map state vectors associated with them respectively." ... "The initial, one-dimensional, two-dimensional, three-dimensional, and cube map texture is therefore operated upon, queried, and applied as TEXTURE_1D, TEXTUER_2D, TEXTURE_3D, and TEXTURE_CUBE_MAP_ARB respectively while 0 is bound to the corresponding targets."

Change paragraph six (page 133) to say:

... "If a texture that is currently bound to one of the targets TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, or TEXTURE_CUBE_MAP_ARB is deleted, it is as though BindTexture has been executed with the same <target> and <texture> zero." ...

-- Section 3.8.10 "Texture Application"

Replace the beginning sentences of the first paragraph (page 138) with:

"Texturing is enabled or disabled using the generic Enable and Disable commands, respectively, with the symbolic constants TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, or TEXTURE_CUBE_MAP_ARB to enable the one-dimensional, two-dimensional, three-dimensional, or cube map texturing respectively. If both two- and one-dimensional textures are enabled, the two-dimensional texture is used. If the three-dimensional and either of the two- or one-dimensional textures is enabled, the three-dimensional texture is used. If the cube map texture and any of the three-, two-, or one-dimensional textures is enabled, then cube map texturing is used. If texturing is disabled, a rasterized fragment is passed on unaltered to the next stage of the GL (although its texture coordinates may be discarded). Otherwise, a texture value is found according to the parameter values of the currently bound texture image of the appropriate dimensionality.

However, when cube map texturing is enabled, the rules are more complicated. For cube map texturing, the (s,t,r) texture coordinates are treated as a direction vector (rx,ry,rz) emanating from the center of a cube. (The q coordinate can be ignored since it merely scales the vector without affecting the direction.) At texture application time, the interpolated per-fragment (s,t,r) selects one of the cube map face's 2D image based on the largest magnitude coordinate direction (the major axis direction). If two or more coordinates have the identical magnitude, the implementation may define the rule to disambiguate this situation. The rule must be deterministic and depend only on (rx,ry,rz). The target column in the table below explains how the major axis direction maps to the 2D image of a particular cube map target.

major axis direction	target	sc	tc	ma
+rx	TEXTURE_CUBE_MAP_POSITIVE_X_ARB	-rz	-ry	rx
-rx	TEXTURE_CUBE_MAP_NEGATIVE_X_ARB	+rz	-ry	rx
+ry	TEXTURE_CUBE_MAP_POSITIVE_Y_ARB	+rx	+rz	ry
-ry	TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB	+rx	-rz	ry
+rz	TEXTURE_CUBE_MAP_POSITIVE_Z_ARB	+rx	-ry	rz
-rz	TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB	-rx	-ry	rz

Using the sc, tc, and ma determined by the major axis direction as specified in the table above, an updated (s,t) is calculated as follows

```
   s   =   ( sc/|ma| + 1 ) / 2
   t   =   ( tc/|ma| + 1 ) / 2
```

This new (s,t) is used to find a texture value in the determined face's 2D texture image using the rules given in sections 3.8.5 and 3.8.6." ...

Additions to Chapter 4 of the 1.2 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.2 Specification (Special Functions)

-- Section 5.4 "Display Lists"

In the first paragraph (page 179), add PROXY_TEXTURE_CUBE_MAP_ARB to the list of PROXY_* tokens.

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

-- Section 6.1.3 "Enumerated Queries"

Change the fourth paragraph (page 183) to say:

"The GetTexParameter parameter <target> may be one of TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, or TEXTURE_CUBE_MAP_ARB, indicating the currently bound one-dimensional, two-dimensional, three-dimensional, or cube map texture object. For GetTexLevelParameter, <target> may be one of TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP_POSITIVE_X_ARB, TEXTURE_CUBE_MAP_NEGATIVE_X_ARB, TEXTURE_CUBE_MAP_POSITIVE_Y_ARB, TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB, TEXTURE_CUBE_MAP_POSITIVE_Z_ARB, TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB, PROXY_TEXTURE_1D, PROXY_TEXTURE_2D, PROXY_TEXTURE_3D, or PROXY_TEXTURE_CUBE_MAP_ARB, indicating the one-dimensional texture object, two-dimensional texture object, three-dimensional texture object, or one of the six distinct 2D images making up the cube map texture object or one-dimensional, two-dimensional, three-dimensional, or cube map proxy state vector. Note that TEXTURE_CUBE_MAP_ARB is not a valid <target> parameter for

    GetTexLevelParameter because it does not specify a particular cube
    map face."

-- Section 6.1.4 "Texture Queries"

    Change the first paragraph (page 184) to read:

    ... "It is somewhat different from the other get commands; <tex>
    is a symbolic value indicating which texture (or texture face in the
    case of a cube map texture target name) is to be obtained.
    TEXTURE_1D indicates a one-dimensional texture, TEXTURE_2D
    indicates a two-dimensional texture, TEXTURE_3D indicates a
    three-dimensional texture, and TEXTURE_CUBE_MAP_POSITIVE_X_ARB,
    TEXTURE_CUBE_MAP_NEGATIVE_X_ARB, TEXTURE_CUBE_MAP_POSITIVE_Y_ARB,
    TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB, TEXTURE_CUBE_MAP_POSITIVE_Z_ARB,
    and TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB indicate the respective face of
    a cube map texture.

Additions to the GLX Specification

    None

Errors

    INVALID_ENUM is generated when TexGen is called with a <coord> of Q
    when <pname> indicates REFLECTION_MAP_ARB or NORMAL_MAP_ARB.

    INVALID_VALUE is generated when the target parameter to TexImage2D
    or CopyTexImage2D is one of the six cube map 2D image targets and
    the width and height parameters are not equal.

New State

(table 6.12, p202) add the following entries:

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
TEXTURE_CUBE_MAP_ARB	B	IsEnabled	False	True if cube map texturing is enabled	3.8.10	texture/enable
TEXTURE_BINDING_CUBE_MAP_ARB	Z+	GetIntegerv	0	Texture object for TEXTURE_CUBE_MAP	3.8.8	texture
TEXTURE_CUBE_MAP_POSITIVE_X_ARB	nxI	GetTexImage	see 3.8	positive x face cube map texture image at lod i	3.8	-
TEXTURE_CUBE_MAP_NEGATIVE_X_ARB	nxI	GetTexImage	see 3.8	negative x face cube map texture image at lod i	3.8	-
TEXTURE_CUBE_MAP_POSITIVE_Y_ARB	nxI	GetTexImage	see 3.8	positive y face cube map texture image at lod i	3.8	-
TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB	nxI	GetTexImage	see 3.8	negative y face cube map texture image at lod i	3.8	-
TEXTURE_CUBE_MAP_POSITIVE_Z_ARB	nxI	GetTexImage	see 3.8	positive z face cube map texture image at lod i	3.8	-
TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB	nxI	GetTexImage	see 3.8	negative z face cube map texture image at lod i	3.8	-

(table 6.14, p204) change the entry for TEXTURE_GEN_MODE to:

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
TEXTURE_GEN_MODE	4xZ5	GetTexGeniv	EYE_LINEAR	Function used for texgen (for s,t,r, and q)	2.10.4	texture

(the type changes from 4xZ3 to 4xZ5)

New Implementation Dependent State

(table 6.24, p214) add the following entry:

Get Value	Type	Get Command	Minimum Value	Description	Sec	Attribute
MAX_CUBE_MAP_TEXTURE_SIZE_ARB	Z+	GetIntegerv	16	Maximum cube map texture image dimension	3.8.1	-

Backwards Compatibility

This extension replaces EXT_texture_cube_map. The tokens and name strings now refer to ARB instead of EXT. Enumerant values are unchanged.

Name

    ARB_texture_env_add

Name Strings

    GL_ARB_texture_env_add

Notice

Status

    Complete. Approved by ARB on 12/8/1999

Version

```
    Last Modified Date: June 22, 2000
    Author Revision: 0.3

    Based on:  EXT_texture_env_add
               Date: 1999/03/22 Revision: 1.1
```

Number

    ARB Extension #6

Dependencies

    None

Overview

```
    New texture environment function ADD is supported with the following
    equation:
                        Cv = min(1, Cf + Ct)

    New function may be specified by calling TexEnv with ADD token.

    One possible application is to add a specular highlight texture to
    a Gouraud-shaded primitive to emulate Phong shading, in a single
    pass.
```

New Procedures and Functions

    None

New Tokens

```
    Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv, and
    TexEnvfi when the <pname> parameter value is GL_TEXTURE_ENV_MODE

        ADD
```

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

None

Additions to Chapter 3 of the GL Specification (Rasterization)

The description of TEXTURE_ENV_MODE in the first paragraph of section 3.8.9 should be modified as follows:

TEXTURE_ENV_MODE may be set to one of REPLACE, MODULATE, DECAL, BLEND or ADD;

Table 3.19 is augmented as follows:

Base Internal Format	DECAL tex func	BLEND tex func	ADD tex func
ALPHA	...	...	Rv = Rf
	...	...	Gv = Gf
	...	...	Bv = Bf
	...	...	Av = AfAt
LUMINANCE	...	...	Rv = min(1, Rf+Lt)
(or 1)	...	...	Gv = min(1, Gf+Lt)
	...	...	Bv = min(1, Bf+Lt)
	...	...	Av = Af
LUMINANCE_ALPHA	...	...	Rv = min(1, Rf+Lt)
(or 2)	...	...	Gv = min(1, Gf+Lt)
	...	...	Bv = min(1, Bf+Lt)
	...	...	Av = AfAt
INTENSITY	...	...	Rv = min(1, Rf+It)
	...	...	Gv = min(1, Gf+It)
	...	...	Bv = min(1, Bf+It)
	...	...	Av = min(1, Af+It)
RGB	...	...	Rv = min(1, Rf+Rt)
(or 3)	...	...	Gv = min(1, Gf+Gt)
	...	...	Bv = min(1, Bf+Bt)
	...	...	Av = Af
RGBA	...	...	Rv = min(1, Rf+Rt)
(or 4)	...	...	Gv = min(1, Gf+Gt)
	...	...	Bv = min(1, Bf+Bt)
	...	...	Av = AfAt

Table 3.19: Decal, blend and add texture functions.

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the GL Specification (Special Functions)

None

Additions to Chapter 6 of the GL Specification (State and State Requests)

None

Additions to the GLX / WGL / AGL Specifications

None

GLX Protocol

None

Errors

None

New State

The Type of TEXTURE_ENV_MODE in Table F.2 should be changed to

```
    1 * xZ5
```

New Implementation Dependent State

None

Revision History

```
11/09/1999  0.1
    - First ARB draft based on the original EXT draft.

1/13/2000   0.2
    - Added justification to the overview
    - Updated to describe modifications to 1.2.1 specification
    - Added changes to description of TEXTURE_ENV_MODE parameter
      to TexEnv{if} and TexEnv{if}v
    - Added change to TEXTURE_ENV_MODE type (Z4 -> Z5)

6/22/2000   0.3
    - The addition should saturate to 1.
```

Name

ARB_texture_env_combine

Name Strings

GL_ARB_texture_env_combine

Version

Last modified date: 2001/05/21

Number

ARB Extension #17

Dependencies

This extension is written against the OpenGL 1.2.1 Specification.
OpenGL 1.1 and ARB_multitexture are required for this extension.

Overview

New texture environment function COMBINE_ARB allows programmable
texture combiner operations, including:

```
    REPLACE                 Arg0
    MODULATE                Arg0 * Arg1
    ADD                     Arg0 + Arg1
    ADD_SIGNED_ARB          Arg0 + Arg1 - 0.5
    SUBTRACT_ARB            Arg0 - Arg1
    INTERPOLATE_ARB         Arg0 * (Arg2) + Arg1 * (1-Arg2)
```

where Arg0, Arg1 and Arg2 are derived from

```
    PRIMARY_COLOR_ARB       primary color of incoming fragment
    TEXTURE                 texture color of corresponding texture unit
    CONSTANT_ARB            texture environment constant color
    PREVIOUS_ARB            result of previous texture environment; on
                            texture unit 0, this maps to PRIMARY_COLOR_ARB
```

In addition, the result may be scaled by 1.0, 2.0 or 4.0.

Issues

1. Should the explicit bias be removed in favor of an implcit bias as
   part of a ADD_SIGNED_ARB function?

   - RESOLVED: Yes. This pre-scale bias is a special case and will
     be treated as such.

2. Should the primary color of the incoming fragment be available to
   all texture environments? Currently it is only available to the
   texture environment of texture unit 0.

   - RESOLVED: Yes. PRIMARY_COLOR_ARB has been added as an input
     source.

3. Should textures from other texture units be allowed as sources?

   - RESOLVED: NO. Even though this adds a lot of flexibility that folks can use today, there is not enough support amonst the ARB participants to add it to the base spec.

4. All of the 1.2 modes except BLEND can be expressed in terms of this extension. Should texture color be allowed as a source for Arg2, so all of the 1.2 modes can be expressed? If so, should all color sources be allowed, to maintain orthogonality?

   - RESOLVED: Yes. This seems to be a reasonable area to expand functionality and remain backwards compatible with the EXT version of the extension.

5. If the texture environment for a given texture unit does not reference the texture object that is bound to that texture unit, does a valid texture object need to be bound that unit?

   - RESOLVED: Yes. Each texture unit implicitly references the texture object that is bound to that unit, regardless of the texture environment function. This may require that applications bind a dummy texture to the texture unit.

6. Should we allow the secondary color to take part in texture blending?

   - RESOLVED: Not in this extension. Secondary color was defined as a specular part of the lit color and does not have associated alpha. In order to do this right, the secondary color extension needs to be fixed first to allow a full featured color and clearly state the interaction of how it interacts with the color sum stage.

7. How exactly is this ARB extension different from the EXT version?

   - RESOLVED:

     1) This extension adds the GL_SUBTRACT_ARB mode

     2) OPERAND2_RGB_ARB can use SRC_COLOR, ONE_MINUS_SRC_COLOR, SRC_ALPHA, and ONE_MINUS_SRC_ALPHA instead of just SRC_ALPHA (NV_texture_env_combine4 already provides this).

     3) OPERAND2_ALPHA_ARB can use SRC_ALPHA and ONE_MINUS_SRC_ALPHA instead of just SRC_ALPHA (NV_texture_env_combine4 already provides this).

New Procedures and Functions

None

New Tokens

Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv, and TexEnviv when the <pname> parameter value is TEXTURE_ENV_MODE

```
    COMBINE_ARB                                     0x8570
```

Accepted by the <pname> parameter of TexEnvf, TexEnvi, TexEnvfv, and TexEnviv when the <target> parameter value is TEXTURE_ENV

COMBINE_RGB_ARB	0x8571
COMBINE_ALPHA_ARB	0x8572
SOURCE0_RGB_ARB	0x8580
SOURCE1_RGB_ARB	0x8581
SOURCE2_RGB_ARB	0x8582
SOURCE0_ALPHA_ARB	0x8588
SOURCE1_ALPHA_ARB	0x8589
SOURCE2_ALPHA_ARB	0x858A
OPERAND0_RGB_ARB	0x8590
OPERAND1_RGB_ARB	0x8591
OPERAND2_RGB_ARB	0x8592
OPERAND0_ALPHA_ARB	0x8598
OPERAND1_ALPHA_ARB	0x8599
OPERAND2_ALPHA_ARB	0x859A
RGB_SCALE_ARB	0x8573
ALPHA_SCALE	

Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv, and TexEnviv when the <pname> parameter value is COMBINE_RGB_ARB or COMBINE_ALPHA_ARB

REPLACE	
MODULATE	
ADD	
ADD_SIGNED_ARB	0x8574
INTERPOLATE_ARB	0x8575
SUBTRACT_ARB	0x84E7

Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv, and TexEnviv when the <pname> parameter value is SOURCE0_RGB_ARB, SOURCE1_RGB_ARB, SOURCE2_RGB_ARB, SOURCE0_ALPHA_ARB, SOURCE1_ALPHA_ARB, or SOURCE2_ALPHA_ARB

TEXTURE	
CONSTANT_ARB	0x8576
PRIMARY_COLOR_ARB	0x8577
PREVIOUS_ARB	0x8578

Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv, and TexEnviv when the <pname> parameter value is OPERAND0_RGB_ARB, OPERAND1_RGB_ARB, or OPERAND2_RGB_ARB

    SRC_COLOR
    ONE_MINUS_SRC_COLOR
    SRC_ALPHA
    ONE_MINUS_SRC_ALPHA

Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv, and TexEnviv when the <pname> parameter value is OPERAND0_ALPHA_ARB, OPERAND1_ALPHA_ARB, or OPERAND2_ALPHA_ARB

    SRC_ALPHA
    ONE_MINUS_SRC_ALPHA

Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv, and TexEnviv when the <pname> parameter value is RGB_SCALE_ARB or ALPHA_SCALE

    1.0
    2.0
    4.0

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

None

Additions to Chapter 3 of the GL Specification (Rasterization)

Added to subsection 3.8.9, before the paragraph describing the state requirements:

If the value of TEXTURE_ENV_MODE is COMBINE_ARB, the form of the texture function depends on the values of COMBINE_RGB_ARB and COMBINE_ALPHA_ARB, according to table 3.20. The RGB and ALPHA results of the texture function are then multiplied by the values of RGB_SCALE_ARB and ALPHA_SCALE, respectively. The results are clamped to [0,1].

COMBINE_RGB_ARB	Texture Function
REPLACE	Arg0
MODULATE	Arg0 * Arg1
ADD	Arg0 + Arg1
ADD_SIGNED_ARB	Arg0 + Arg1 - 0.5
INTERPOLATE_ARB	Arg0 * (Arg2) + Arg1 * (1-Arg2)
SUBTRACT_ARB	Arg0 - Arg1

COMBINE_ALPHA_ARB	Texture Function
REPLACE	Arg0
MODULATE	Arg0 * Arg1
ADD	Arg0 + Arg1
ADD_SIGNED_ARB	Arg0 + Arg1 - 0.5
INTERPOLATE_ARB	Arg0 * (Arg2) + Arg1 * (1-Arg2)
SUBTRACT_ARB	Arg0 - Arg1

Table 3.20: COMBINE_ARB texture functions

The arguments Arg0, Arg1 and Arg2 are determined by the values of SOURCE<n>_RGB_ARB, SOURCE<n>_ALPHA_ARB, OPERAND<n>_RGB_ARB and OPERAND<n>_ALPHA_ARB. In the following two tables, Ct and At are the filtered texture RGB and alpha values; Cc and Ac are the texture environment RGB and alpha values; Cf and Af are the RGB and alpha of the primary color of the incoming fragment; and Cp and Ap are the RGB and alpha values resulting from the previous texture environment. On texture environment 0, Cp and Ap are identical to Cf and Af, respectively. The relationship is described in tables 3.21 and 3.22.

SOURCE<n>_RGB_ARB	OPERAND<n>_RGB_ARB	Argument
TEXTURE	SRC_COLOR	Ct
	ONE_MINUS_SRC_COLOR	(1-Ct)
	SRC_ALPHA	At
	ONE_MINUS_SRC_ALPHA	(1-At)
CONSTANT_ARB	SRC_COLOR	Cc
	ONE_MINUS_SRC_COLOR	(1-Cc)
	SRC_ALPHA	Ac
	ONE_MINUS_SRC_ALPHA	(1-Ac)
PRIMARY_COLOR_ARB	SRC_COLOR	Cf
	ONE_MINUS_SRC_COLOR	(1-Cf)
	SRC_ALPHA	Af
	ONE_MINUS_SRC_ALPHA	(1-Af)
PREVIOUS_ARB	SRC_COLOR	Cp
	ONE_MINUS_SRC_COLOR	(1-Cp)
	SRC_ALPHA	Ap
	ONE_MINUS_SRC_ALPHA	(1-Ap)

Table 3.21: Arguments for COMBINE_RGB_ARB functions

SOURCE<n>_ALPHA_ARB	OPERAND<n>_ALPHA_ARB	Argument
TEXTURE	SRC_ALPHA	At
	ONE_MINUS_SRC_ALPHA	(1-At)
CONSTANT_ARB	SRC_ALPHA	Ac
	ONE_MINUS_SRC_ALPHA	(1-Ac)
PRIMARY_COLOR_ARB	SRC_ALPHA	Af
	ONE_MINUS_SRC_ALPHA	(1-Af)
PREVIOUS_ARB	SRC_ALPHA	Ap
	ONE_MINUS_SRC_ALPHA	(1-Ap)

Table 3.22: Arguments for COMBINE_ALPHA_ARB functions

The mapping of texture components to source components is summarized in Table 3.23. In the following table, At, Lt, It, Rt, Gt and Bt are the filtered texel values.

Base Internal Format	RGB Values	Alpha Value
ALPHA	0, 0, 0	At
LUMINANCE	Lt, Lt, Lt	1
LUMINANCE_ALPHA	Lt, Lt, Lt	At
INTENSITY	It, It, It	It
RGB	Rt, Gt, Bt	1
RGBA	Rt, Gt, Bt	At

Table 3.23: Correspondence of texture components to source components for COMBINE_RGB_ARB and COMBINE_ALPHA_ARB arguments

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the GL Specification (Special Functions)

None

Additions to Chapter 6 of the GL Specification (State and State Requests)

None

Additions to Appendix F of the GL Specification (ARB Extensions)

Inserted after the second paragraph of F.2.12:

If the value of TEXTURE_ENV_MODE is COMBINE_ARB, the texture function associated with a given texture unit is computed using the values specified by SOURCE<n>_RGB_ARB, SOURCE<n>_ALPHA_ARB, OPERAND<n>_RGB_ARB and OPERAND<n>_ALPHA_ARB. If TEXTURE<n>_ARB is specified as SOURCE<n>_RGB_ARB or SOURCE<n>_ALPHA_ARB, the texture value from texture unit <n> will be used in computing the texture function for this texture unit.

Inserted after the third paragraph of F.2.12:

If a texture environment for a given texture unit references a texture unit that is disabled or does not have a valid texture object bound to it, then it is as if texture is disabled for the given texture unit. Every texture unit implicitly references the texture object that is bound to it, regardless of the texture function specified by COMBINE_RGB_ARB or COMBINE_ALPHA_ARB.

Additions to the GLX Specification

None

GLX Protocol

None

Errors

INVALID_ENUM is generated if <params> value for COMBINE_RGB_ARB or COMBINE_ALPHA_ARB is not one of REPLACE, MODULATE, ADD, ADD_SIGNED_ARB, INTERPOLATE_ARB, or SUBTRACT_ARB

INVALID_ENUM is generated if <params> value for SOURCE0_RGB_ARB, SOURCE1_RGB_ARB, SOURCE2_RGB_ARB, SOURCE0_ALPHA_ARB, SOURCE1_ALPHA_ARB or SOURCE2_ALPHA_ARB is not one of TEXTURE, CONSTANT_ARB, PRIMARY_COLOR_ARB, or PREVIOUS_ARB.

INVALID_ENUM is generated if <params> value for OPERAND0_RGB_ARB, OPERAND1_RGB_ARB, or OPERAND2_RGB_ARB is not one of SRC_COLOR, ONE_MINUS_SRC_COLOR, SRC_ALPHA or ONE_MINUS_SRC_ALPHA.

INVALID_ENUM is generated if <params> value for OPERAND0_ALPHA_ARB, OPERAND1_ALPHA_ARB, or OPERAND2_ALPHA_ARB is not one of SRC_ALPHA or ONE_MINUS_SRC_ALPHA.

INVALID_VALUE is generated if <params> value for RGB_SCALE_ARB or ALPHA_SCALE is not one of 1.0, 2.0, or 4.0.

New State

Get Value	Get Command	Type	Initial Value	Attribute
COMBINE_RGB_ARB	GetTexEnviv	n x Z4	MODULATE	texture
COMBINE_ALPHA_ARB	GetTexEnviv	n x Z4	MODULATE	texture
SOURCE0_RGB_ARB	GetTexEnviv	n x Z3	TEXTURE	texture
SOURCE1_RGB_ARB	GetTexEnviv	n x Z3	PREVIOUS_ARB	texture
SOURCE2_RGB_ARB	GetTexEnviv	n x Z3	CONSTANT_ARB	texture
SOURCE0_ALPHA_ARB	GetTexEnviv	n x Z3	TEXTURE	texture
SOURCE1_ALPHA_ARB	GetTexEnviv	n x Z3	PREVIOUS_ARB	texture
SOURCE2_ALPHA_ARB	GetTexEnviv	n x Z3	CONSTANT_ARB	texture
OPERAND0_RGB_ARB	GetTexEnviv	n x Z6	SRC_COLOR	texture
OPERAND1_RGB_ARB	GetTexEnviv	n x Z6	SRC_COLOR	texture
OPERAND2_RGB_ARB	GetTexEnviv	n x Z1	SRC_ALPHA	texture
OPERAND0_ALPHA_ARB	GetTexEnviv	n x Z4	SRC_ALPHA	texture
OPERAND1_ALPHA_ARB	GetTexEnviv	n x Z4	SRC_ALPHA	texture
OPERAND2_ALPHA_ARB	GetTexEnviv	n x Z1	SRC_ALPHA	texture
RGB_SCALE_ARB	GetTexEnvfv	n x R3	1.0	texture
ALPHA_SCALE	GetTexEnvfv	n x R3	1.0	texture

New Implementation Dependent State

None

Revision History

01/05/21	mjk	Added ARB versus EXT differences issue
01/00/02	bpoddar	Added original EXT/ARB contributors to the contact list
00/12/13	bpoddar	Added enum value for SUBTRACT_ARB
00/12/06	bpoddar	Moved references to Ct<n> and At<n> to ARB_texture_env_crossbar spec.
00/12/01	bpoddar	Removed TEXTURE<n>_ARB since several companies had problems with this addition in the base spec.
00/11/13	bpoddar	Recreated 6/20 spec with language for dealing with inconsistent textures moved to appendix F.
00/06/20	rhammers	Changed behavior when dealing with references do disabled and inconsistent textures.
00/05/23	rhammers	Cleaned up for first draft of ARB version. Added issue -- TEXTURE with TEXTURE<n>_ARB Added issue .. "upstream" textures Listed get functions with description of enumerants. Added 1.1 and multitexture to dependencies

```
    00/05/18  rhammers  First rev of ARB version of the spec. Based on
                        EXT_texture_env_combine.
                        Relaxed restriction on Arg2.
                        Added support for TEXTURE<n>_ARB.
                        Added SUBTRACT_ARB combiner function.
                        do disabled and inconsistent textures.
```

Name

    ARB_texture_env_crossbar

Name Strings

    GL_ARB_texture_env_crossbar

Overview

This extension adds the capability to use the texture color from other texture units as sources to the COMBINE_ARB enviornment function. The ARB_texture_env_combine extension defined texture enviornment functions which could use the color from the current texture unit as a source. This extension adds the ability to use the color from any texture unit as a source.

NVIDIA Note

The NV_texture_env_combine4 extension provides nearly identical functionality to functionality that the ARB_texture_env_crossbar extension provides.

Unfortunately, the ARB_texture_env_crossbar's semantic for what happens when a texture environment stage references a disabled texture does not match NVIDIA's NV_texture_env_combine behavior. Due to the differing semantics and in order to maintain backward application compatibility and compatibility with the NV_texture_env_combine4 specification, NVIDIA will ***never*** advertise the ARB_texture_env_crossbar extension.

The ARB_texture_env_combine semantic is:

> Texture blending should be disabled on the texture unit that is referencing the invalid or disabled texture.

The NV_texture_env_combine4 semantic is:

> If the <n>th texture unit is disabled, the value of each component is 1.

Fortunately, this semantic is not particularly relevant for most applications because applications typically avoid sourcing a disabled, inconsistent, or invalid texture unit.

We recommend that if your application sources other texture units using the GL_COMBINE_ARB texture envionment mode, you first determine that ***either*** ARB_texture_env_crossbar ***or*** NV_texture_env_combine4 are supported. Then do not assume a particular behavior when sourcing other texture units with GL_COMBINE_ARB environment that are disabled or invalid.

OpenGL 1.4 codifies this practice by integrating the ARB_texture_env_crossbar functionality into the core OpenGL standard. The OpenGL 1.4 standard says: "If the texture environment for a given enabled texture unit references a disabled texture unit,

```
    or an invalid or incomplete texture that is bound to another unit,
    then the result of texture blending are undefined."
```

Web Reference

```
    http://oss.sgi.com/projects/ogl-sample/registry/ARB/texture_env_crossbar.txt
```

Name

    ARB_texture_env_dot3

Name Strings

    GL_ARB_texture_env_dot3

Status

    Complete. Approved by ARB on February 16, 2001.

Version

    Last modified date: 2001/05/16

Number

    ARB Extension #19

Dependencies

```
    This extension is written against the OpenGL 1.2.1 Specification.
    OpenGL 1.1, ARB_multitexture and ARB_texture_env_combine are required
    for this extension.
```

Overview

```
    Adds new operation to the texture combiner operations.

        DOT3_RGB_ARB                    Arg0 <dotprod> Arg1
        DOT3_RGBA_ARB                   Arg0 <dotprod> Arg1

    where Arg0, Arg1 are specified by <params> parameter of
    TexEnvf, TexEnvi, TexEnvfv, and TexEnviv when the <pname>
    parameter value is SOURCE0_RGB_ARB and SOURCE1_RGB_ARB.
```

Issues

```
 1. This extension is an ARB version of EXT_texture_env_dot3 which bears
    a copyright by ATI Technologies. Is ATI willing to have the ARB
    go ahead and modify their original spec and use it for the
    ARB extension.

    - RESOLVED: ATI does not have a problem with the copyright issue.

 2. The EXT version of the spec does not multiply the output by
    RGB_SCALE_ARB and ALPHA_SCALE_ARB. There is no reason to impose this
    restriction since it makes the scale operations non-orthogonal.
    Should the enum values for the new tokens in this extension should
    be the same as the original EXT version?

    - RESOLVED: No.

 3. How exactly is this ARB extension different from the EXT version?

    - RESOLVED:  Scaling by 2.0 and 4.0 is supported by the ARB version,
      but not the EXT version (as noted above).  Note that when
      DOT3_RGBA_ARB is used, the alpha component result is scaled
      based on the RGB scale factor rather than the alpha scale factor
      (the COMBINE_ALPHA_ARB function and scale factor are ignored).
      The COMBINE_ALPHA_ARB mode is ignored in the EXT version and the
      previous alpha is passed through; however, the ARB version abides
      by the COMBINE_ALPHA_ARB setting.
```

New Procedures and Functions

```
    None
```

New Tokens

```
    Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv,
    and TexEnviv when the <pname> parameter value is COMBINE_RGB_ARB

        DOT3_RGB_ARB                                    0x86AE
        DOT3_RGBA_ARB                                   0x86AF
```

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

```
    None
```

Additions to Chapter 3 of the GL Specification (Rasterization)

```
    Added to table 3.20 of the ARB_texture_env_combine spec:

        COMBINE_RGB_ARB         Texture Function
        ---------------         ----------------
        DOT3_RGB_ARB            4*((Arg0_r - 0.5)*(Arg1_r - 0.5) +
                                   (Arg0_g - 0.5)*(Arg1_g - 0.5) +
                                   (Arg0_b - 0.5)*(Arg1_b - 0.5))

                                This value is placed into all three
                                r,g,b components of the output.

        DOT3_RGBA_ARB           4*((Arg0_r - 0.5)*(Arg1_r - 0.5) +
                                   (Arg0_g - 0.5)*(Arg1_g - 0.5) +
                                   (Arg0_b - 0.5)*(Arg1_b - 0.5))

                                This value is placed into all four
                                r,g,b,a components of the output. Note
                                that the result generated from
                                COMBINE_ALPHA_ARB function is ignored.
```

Additions to Chapter 4 of the OpenGL 1.2 Specification (Per-Fragment Operations and the Framebuffer)

```
    None
```

Additions to Chapter 5 of the GL Specification (Special Functions)

```
    None
```

Additions to Chapter 6 of the GL Specification (State and State Requests)

    None

Additions to the GLX Specification

    None

GLX Protocol

    None

Errors

    INVALID_ENUM is generated if <params> value for COMBINE_RGB_ARB
    is not one of REPLACE, MODULATE, ADD, ADD_SIGNED_ARB,
    INTERPOLATE_ARB, SUBTRACT_ARB, DOT3_RGB_ARB or DOT3_RGBA_ARB.

New State

    None

New Implementation Dependent State

    None

Revision History

```
    01/05/16  mjk       Dot3 combiner operations not allowed for alpha portion
    01/02/02  bpoddar   Added original EXT/ARB contributors to the contact
                        list

    00/12/13  bpoddar   Added enum values for DOT3_RGB_ARB and DOT3_RGBA_ARB
                        Added resolution to issue # 1.

    00/12/06  bpoddar   Fixed typos - EXT -> ARB, RED_SCALE -> RGB_SCALE

    00/12/01  bpoddar   Created an ARB version of the ARB_texture_env_dot3
                        by breaking up the proposed ARB_texture_env_combine
                        spec.
```

Name

    ARB_texture_float

Name Strings

    GL_ARB_texture_float

Contributors

    Pat Brown
    Jon Leech
    Rob Mace
    Brian Paul

Contact

    Dale Kirkland, NVIDIA (dkirkland 'at' nvidia.com)

Status

    Complete. Appprove by the ARB on October 22, 2004.

Version

    Based on the ATI_texture_float extension, verion 4

    Last Modified Date: July 6, 2006
    Version: 6

Number

    ARB Extension #41

Dependencies

    This extension is written against the OpenGL 2.0 Specification
    but will work with the OpenGL 1.5 Specification.

    OpenGL 1.1 or EXT_texture is required.

    This extension interacts with ARB_color_buffer_float.

Overview

    This extension adds texture internal formats with 16- and 32-bit
    floating-point components. The 32-bit floating-point components
    are in the standard IEEE float format. The 16-bit floating-point
    components have 1 sign bit, 5 exponent bits, and 10 mantissa bits.
    Floating-point components are clamped to the limits of the range
    representable by their format.

IP Status

    SGI owns US Patent #6,650,327, issued November 18, 2003. SGI
    believes this patent contains necessary IP for graphics systems
    implementing floating point (FP) rasterization and FP framebuffer

capabilities.

SGI will not grant the ARB royalty-free use of this IP for use in OpenGL, but will discuss licensing on RAND terms, on an individual basis with companies wishing to use this IP in the context of conformant OpenGL implementations. SGI does not plan to make any special exemption for open source implementations.

Contact Doug Crisman at SGI Legal for the complete IP disclosure.

Issues

1. *How is this extension different from the ATI_texture_float extension?*

   This extension expands on the definition of float16 values and adds a query to determine if the components of a texture are stored as floats.

2. *Should the new names of the internal formats be changed to a different spelling?*

   RESOLVED: Internal format names have been updated to the same convention as the EXT_framebuffer_object extension.

3. *Is it allowable for an implementation to fall back to a non floating-point internal format if it does not support the requested format?*

   RESOLVED: No. An application that requests floating-point formats should expect to get them. Only the precision of the internal format can be changed. When this extension is promoted to the core, this issue may need to be readdressed.

4. *Do the new internal formats apply to any other commands?*

   RESOLVED: Since color tables support the same <internalFormat> values as textures, they are also extended with this extension, except the individual component types cannot be queried.

5. *Are the floating-point values clamped before they are stored into the texture memory or color tables?*

   RESOLVED: The values are clamped to the representatable range of the storage format. Overflows could produce +/-INF and underflows could produce denorms or zero. This matches the behavior of the ATI extension.

6. *Should this extension modify the clamping of the texture border color components?*

   RESOLVED: Yes. The border color components are unclamped. When used, the border color components are interpreted in a manner consistent with the texture's internal format. For fixed-point textures, this means that the border color is clamped to [0, 1] when used.

7. Are floating-point values clamped for the fixed-function GL?

RESOLVED: This extension introduces texel values that can be outside [0, 1]. No clamping occurs to these values during texture filtering. For the fixed-function pipeline, the filtered texel is now clamped before it is used for texture environment blending. The ARB_color_buffer_float extension can be used to control this clamping. For the programmable pipelines, no clamping occurs.

8. Should the query for the border color return the unclamped values?

RESOLVED: There is language in the ARB_color_buffer_float extension that handles this. Since there is no clamp control in this specification, it would be hard to do anything other than return the clamped values.

New Procedures and Functions

None

New Tokens

Accepted by the <value> parameter of GetTexLevelParameter:

TEXTURE_RED_TYPE_ARB	0x8C10
TEXTURE_GREEN_TYPE_ARB	0x8C11
TEXTURE_BLUE_TYPE_ARB	0x8C12
TEXTURE_ALPHA_TYPE_ARB	0x8C13
TEXTURE_LUMINANCE_TYPE_ARB	0x8C14
TEXTURE_INTENSITY_TYPE_ARB	0x8C15
TEXTURE_DEPTH_TYPE_ARB	0x8C16

Returned by the <params> parameter of GetTexLevelParameter:

UNSIGNED_NORMALIZED_ARB	0x8C17

Accepted by the <internalFormat> parameter of TexImage1D, TexImage2D, and TexImage3D:

RGBA32F_ARB	0x8814
RGB32F_ARB	0x8815
ALPHA32F_ARB	0x8816
INTENSITY32F_ARB	0x8817
LUMINANCE32F_ARB	0x8818
LUMINANCE_ALPHA32F_ARB	0x8819
RGBA16F_ARB	0x881A
RGB16F_ARB	0x881B
ALPHA16F_ARB	0x881C
INTENSITY16F_ARB	0x881D
LUMINANCE16F_ARB	0x881E
LUMINANCE_ALPHA16F_ARB	0x881F

Additions to Chapter 2 of the OpenGL 2.0 Specification (OpenGL Operation)

Add a new Section 2.1.2, (p. 6):

2.1.2 16-Bit Floating-Point Numbers

A 16-bit floating-point number has a 1-bit sign (S), a 5-bit exponent (E), and a 10-bit mantissa (M). The value of a 16-bit floating-point number is determined by the following:

```
    (-1)^S * 0.0,                        if E == 0 and M == 0,
    (-1)^S * 2^-14 * (M / 2^10),         if E == 0 and M != 0,
    (-1)^S * 2^(E-15) * (1 + M/2^10),    if 0 < E < 31,
    (-1)^S * INF,                        if E == 31 and M == 0, or
    NaN,                                 if E == 31 and M != 0,
```

where

```
    S = floor((N mod 65536) / 32768),
    E = floor((N mod 32768) / 1024), and
    M = N mod 1024.
```

Implementations are also allowed to use any of the following alternative encodings:

```
    (-1)^S * 0.0,                        if E == 0 and M != 0,
    (-1)^S * 2^(E-15) * (1 + M/2^10),    if E == 31 and M == 0, or
    (-1)^S * 2^(E-15) * (1 + M/2^10),    if E == 31 and M != 0,
```

Any representable 16-bit floating-point value is legal as input to a GL command that accepts 16-bit floating-point data. The result of providing a value that is not a floating-point number (such as infinity or NaN) to such a command is unspecified, but must not lead to GL interruption or termination. Providing a denormalized number or negative zero to GL must yield predictable results.

Additions to Chapter 3 of the OpenGL 2.0 Specification (Rasterization)

Modify Section 3.6.3 (Pixel Transfer Modes), p. 116

(modify first paragraph, p. 118) The specified image is taken from memory and processed just as if DrawPixels were called, stopping after the final expansion to RGBA. The R, G, B, and A components of each pixel are then scaled by the four COLOR TABLE SCALE parameters and biased by the four COLOR TABLE BIAS parameters. These parameters are set by calling ColorTableParameterfv as described below. If fragment color clamping is enable or the <internalformat> is fixed-point, the components are clamped to [0, 1]. Otherwise, the components are not modified.

Modify Section 3.8.1 (Texture Image Specification), p. 150

(modify second paragraph, p. 151) The selected groups are processed exactly as for DrawPixels, stopping just before final conversion. For R, G, B, and A, if the <internalformat> of the texture is fixed-point, the components are clamped to [0, 1]. Otherwise, the components are not modified. The depth value so generated is clamped to [0, 1].

(modify the second paragraph, p. 152) The internal component resolution is the number of bits allocated to each value in a texture image. If <internalformat> is specified as a base internal format, the GL stores the resulting texture with internal component resolutions of its own choosing. If a sized internal format is specified, the mapping of the R, G, B, A, and depth values to texture components is equivalent to the mapping of the corresponding base internal format's components, as specified in table 3.15, the type (unsigned int, float, etc.) is assigned the same type specified by <internalFormat>, and the memory allocation per texture component is assigned by the GL to match the allocations listed in table 3.16 as closely as possible. (The definition of closely is left up to the implementation. Implementations are not required to support more than one resolution of each type (unsigned int, float, etc.) for each base internal format.) If a compressed internal format is specified, the mapping of the R, G, B, A, and depth values to texture components is equivalent to the mapping of the corresponding base internal format's components, as specified in table 3.15. The specified image is compressed using a (possibly lossy) compression algorithm chosen by the GL.

(add the following to table 3.16, p. 154)

Sized Internal Format	Base Internal Format	R bits	G bits	B bits	A bits	L bits	I bits
RGBA32F_ARB	RGBA	f32	f32	f32	f32		
RGB32F_ARB	RGB	f32	f32	f32			
ALPHA32F_ARB	ALPHA				f32		
INTENSITY32F_ARB	INTENSITY						f32
LUMINANCE32F_ARB	LUMINANCE					f32	
LUMINANCE_ALPHA32F_ARB	LUMINANCE_ALPHA				f32	f32	
RGBA16F_ARB	RGBA	f16	f16	f16	f16		
RGB16F_ARB	RGB	f16	f16	f16			
ALPHA16F_ARB	ALPHA				f16		
INTENSITY16F_ARB	INTENSITY						f16
LUMINANCE16F_ARB	LUMINANCE					f16	
LUMINANCE_ALPHA16F_ARB	LUMINANCE_ALPHA				f16	f16	

Table 3.16: Correspondence of sized internal formats to base internal formats, and desired component resolutions for each sized internal format. The notation <f16> and <f32> imply 16- and 32-bit floating-point, respectively.

Modify Section 3.8.4 (Texture Parameters), p. 166

(remove TEXTURE_BORDER_COLOR from end of first paragraph, p. 166)

... If the values for TEXTURE_BORDER_COLOR or the value for TEXTURE_PRIORITY are specified as integers, the conversion for signed integers from table 2.9 is applied to convert this value to floating-point. Regardless of the original data type, the value for TEXTURE_PRIORITY is clamped to lie in [0, 1].

... If the value for TEXTURE_PRIORITY is specified as an integer, the conversion for signed integers from table 2.9 is applied to convert this value to floating-point, followed by clamping the value to lie in [0, 1].

Modify Section 3.8.8 (Texture Minification), p. 170

(modify last paragraph, p. 174) ... If the texture contains color components, the values of TEXTURE BORDER COLOR are interpreted as an RGBA color to match the texture's internal format in a manner consistent with table 3.15. The border values for texture components stored as fixed-point values are clamped to [0, 1] before they are used. If the texture contains depth ...

Modify Section 3.8.11 (Texture State and Proxy State) p. 178

(modify the first section, p. 178) ...Each array has associated with it a width, height (two- and three-dimensional and cubemap only), and depth (three-dimensional only), a border width, an integer describing the internal format of the image, six integer values describing the resolutions of each of the red, green, blue, alpha, luminance, and intensity components of the image, six values that describe the type (unsigned int, floats, etc.) of each of the red, green, blue, alpha, luminance, and intensity components of the image, a boolean describing whether the image is compressed or not, and an integer size of a compressed image. Each initial...

(modify the first paragraph, p. 179) ...Each proxy array includes width, height (two- and three- dimensional arrays only), depth (three-dimensional arrays only), border width, and internal format state values, as well as state for the red, green, blue, alpha, luminance, and intensity component resolutions and types (unsigned int, floats, etc.). Proxy arrays do not include image data, nor do they include texture properties. When TexImage3D is executed with target specified as PROXY TEXTURE 3D, the three-dimensional proxy state values of the specified level-of-detail are recomputed and updated. If the image array would not be supported by TexImage3D called with target set to TEXTURE 3D, no error is generated, but the proxy width, height, depth, border width, and component resolutions are set to zero, and the component types are set to NONE. If the image...

Modify Section 3.8.13 (Texture Environments and Functions), p.182

(replace the sixth paragraph of p. 183) All of these color values are clamped to the range [0, 1]. The texture functions are specified in tables 3.22, 3.23, and 3.24.

Additions to Chapter 4 of the OpenGL 2.0 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the OpenGL 2.0 Specification (Special Functions)

None

Additions to Chapter 6 of the OpenGL 2.0 Specification (State and State Requests)

Modify Section 6.1.3 (Enumerated Queries), p. 246

(modify second paragraph, p. 247) For texture images with uncompressed internal formats, queries of <value> of TEXTURE_RED_TYPE_ARB, TEXTURE_GREEN_TYPE_ARB, TEXTURE_BLUE_TYPE_ARB, TEXTURE_ALPHA_TYPE_ARB, TEXTURE_LUMINANCE_TYPE_ARB, TEXTURE_INTENSITY_TYPE_ARB, and TEXTURE_DEPTH_TYPE_ARB, return either NONE, UNSIGNED_NORMALIZED_ARB, or FLOAT indicating how the components are stored, and the queries of <value> of TEXTURE_RED_SIZE, TEXTURE_GREEN_SIZE, TEXTURE_BLUE_SIZE, TEXTURE_ALPHA_SIZE, TEXTURE_LUMINANCE_SIZE, TEXTURE_DEPTH_SIZE, and TEXTURE_INTENSITY_SIZE return the actual resolutions of the stored image array components, not the resolutions specified when the image array was defined.

Additions to the AGL/GLX/WGL Specifications

None

Dependencies on ARB_color_buffer_float extension

The ARB_color_buffer_float extension allows clamping to be controlled in various parts of the GL. Specifically, clamping of filtered texel values used for texture environment blending can be disable.

Errors

None

New State

(Table 6.17, p. 278) add the following entries:

Get Value	Type	Get Command	Minimum Value	Description	Section	Attribute
TEXTURE_RED_TYPE_ARB	Z3	GetTexLevelParameter	-	storage type	6.1.3	-
TEXTURE_GREEN_TYPE_ARB	Z3	GetTexLevelParameter	-	storage type	6.1.3	-
TEXTURE_BLUE_TYPE_ARB	Z3	GetTexLevelParameter	-	storage type	6.1.3	-
TEXTURE_ALPHA_TYPE_ARB	Z3	GetTexLevelParameter	-	storage type	6.1.3	-
TEXTURE_LUMINANCE_TYPE_ARB	Z3	GetTexLevelParameter	-	storage type	6.1.3	-
TEXTURE_INTENSITY_TYPE_ARB	Z3	GetTexLevelParameter	-	storage type	6.1.3	-
TEXTURE_DEPTH_TYPE_ARB	Z3	GetTexLevelParameter	-	storage type	6.1.3	-

New Implementation Dependent State

None

Revision History

Rev.	Date	Author	Changes
1	2/26/04	kirkland	Initial version based on the ATI extension.

```
    2   3/11/04  kirkland   Updated language for float16 number
                            handling.
                            Added bit encodings for half values.
                            Added an issue for color tables.
                            Added separate queries for component
                            types.
                            Changed the internal format names to
                            match the uber buffer extension.
                            Added language to not allow textures to
                            change the type of the internal formats,
                            only the precision.

    3   7/23/04  kirkland   Added alternative encodings options for
                            float16 format.

    4   9/17/04  kirkland   Updated to reference the OpenGL 2.0 spec.
                            Added interaction with clamp control.
                            Removed the clamping of color table data.

    5   10/1/04  Kirkland   Updated IP section.
                            Reviewed by the ARB and closed all
                            UNRESOLVED issues.

    6    7/6/06  pbrown     Fixed broken language for border color
                            handling.  TexParameteriv border colors
                            should still be converted to integer; we
                            only intended to remove the [0,1] clamping.
```

Name

    ARB_texture_mirrored_repeat

Name Strings

    GL_ARB_texture_mirrored_repeat

Status

    Complete. Approved by ARB on October 16, 2001.

Version

    Last modified date: 2001/09/20

Number

    ARB Extension #21

Dependencies

    This extension is written against the OpenGL 1.3 Specification.
    However, this extension does not require OpenGL 1.3.

Overview

    ARB_texture_mirrored_repeat extends the set of texture wrap modes to
    include a mode (GL_MIRRORED_REPEAT_ARB) that effectively uses a texture
    map twice as large at the original image in which the additional half,
    for each coordinate, of the new image is a mirror image of the original
    image.

    This new mode relaxes the need to generate images whose opposite edges
    match by using the original image to generate a matching "mirror image".

Issues

     1. The spec clamps the final (u,v) coordinates to the range [0.5, 2^n-0.5].
        This will produce the same effect as trapping a sample of the border texel
        and using the corresponding edge texel. The choice of technique is purely
        an implementation detail.

     2. The IBM_texture_mirrored_repeat extension inadvertantly used an HP
        enumerant value (0x8370) allocated by HP as an interleaved array format.
        Should the enumerant value be changed if this becomes an ARB extension?

        No, it is not worth the confusion created by having two different
        enumerant value for the same token.

     3. Should additional mirroring functions be added to this extension and
        perhaps rename it to ARB_texture_mirror. For example, include the two
        mirror modes (MIRROR_CLAMP_ATI and MIRROR_CLAMP_TO_EDGE_ATI) provided
        for in the GL_ATI_texture_mirror_once extension.

    No, these extensions are not interdependent and inclusion of the
    mirror once will likely hinder the adoption of this extension.

New Procedures and Functions

    None

New Tokens

    Accepted by the <param> parameter of TexParameteri and TexParameterf,
    and by the <params> parameter of TexParameteriv and TexParameterfv, when
    their <pname> parameter is TEXTURE_WRAP_S, TEXTURE_WRAP_T, or
    TEXTURE_WRAP_R:

```
      GL_MIRRORED_REPEAT_ARB                   0x8370
```

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

    None.

Additions to Chapter 3 of the GL Specification (Rasterization)

  Modify Table 3.19, editing only the following lines:

```
    Name              Type      Legal Values
    ==============    =======   ====================
    TEXTURE_WRAP_S    integer   CLAMP, CLAMP_TO_EDGE, REPEAT,
                                CLAMP_TO_BORDER_ARB, MIRRORED_REPEAT_ARB
    TEXTURE_WRAP_T    integer   CLAMP, CLAMP_TO_EDGE, REPEAT,
                                CLAMP_TO_BORDER_ARB, MIRRORED_REPEAT_ARB
    TEXTURE_WRAP_R    integer   CLAMP, CLAMP_TO_EDGE, REPEAT,
                                CLAMP_TO_BORDER_ARB, MIRRORED_REPEAT_ARB
```

  Add to end of Section 3.8.5 (Subsection "Texture Wrap Modes")

    If TEXTURE_WRAP_S, TEXTURE_WRAP_T, or TEXTURE_WRAP_R is set to
    MIRRORED_REPEAT_ARB, the s (or t or r) coordinate is converted to:

```
        s - floor(s),           if floor(s) is even, or
        1 - (s - floor(s)),     if floor(s) is odd.
```

    The converted s (or t or r) coordinate is then clamped
    as described for CLAMP_TO_EDGE texture coordinate clamping.

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

    None

Additions to Chapter 5 of the GL Specification (Special Functions)

    None

Additions to Chapter 6 of the GL Specification (State and State Requests)

    None

Additions to Appendix F of the GL Specification (ARB Extensions)

None

Additions to the GLX Specification

None

GLX Protocol

None.

Errors

None

New State

Only the type information changes for these parameters:

```
                                          Initial
Get Value        Get Command      Type    Value   Description        Sec.   Attrib
---------        -----------      ----    ------- -----------        ----   ------
TEXTURE_WRAP_S   GetTexParameteriv   n x Z5 REPEAT  Texture Wrap Mode S  3.8    texture
TEXTURE_WRAP_T   GetTexParameteriv   n x Z5 REPEAT  Texture Wrap Mode T  3.8    texture
TEXTURE_WRAP_R   GetTexParameteriv   n x Z5 REPEAT  Texture Wrap Mode R  3.8    texture
```

New Implementation Dependent State

None

Revision History

```
    01/09/20 bpoddar    - Moved description for section 3.8.5 to the end
                          to avoid a forward reference
                        - Changed to using the old enumerant
                        - Minor typo/email address fixes

    01/09/11 bpoddar    - Updated for OpenGL 1.3 spec.
                        - Minor change to description of clamping.

    01/03/22 brokensh   Converted the IBM extension to a ARB extension
                        written against the latest specification.
```

Name

    ARB_texture_non_power_of_two

Name Strings

    GL_ARB_texture_non_power_of_two

Notice

    Copyright to be assigned to the ARB.

Status

    Approved by the ARB on June 11, 2003.

Version

    Date: May 14, 2004
    Revision: 1.0

Number

    ARB Extension #34

Dependencies

    Written based on the OpenGL 1.4 specification.

    ARB_texture_mirrored_repeat (and IBM_texture_mirrored_repeat)
    affects the definition of this extension.

    ARB_texture_border_clamp affects the definition of this extension.

    EXT_texture_compression_s3tc and NV_texture_compression_vtc affect
    the definition of this extension.

Overview

    Conventional OpenGL texturing is limited to images with
    power-of-two dimensions and an optional 1-texel border.
    ARB_texture_non_power_of_two extension relaxes the size restrictions
    for the 1D, 2D, cube map, and 3D texture targets.

    There is no additional procedural or enumerant api introduced by this
    extension except that an implementation which exports the extension
    string will allow an application to pass in texture dimensions for
    the 1D, 2D, cube map, and 3D targets that may or may not be a power
    of two.

    An implementation which supports relaxing traditional GL's
    power-of-two size restrictions across all texture targets will export
    the extension string: "ARB_texture_non_power_of_two".

    When this extension is supported, mipmapping, automatic mipmap
    generation, and all the conventional wrap modes are supported for
    non-power-of-two textures

Issues

1. What should this extension be called?

STATUS: RESOLVED

RESOLUTION: ARB_texture_non_power_of_two. Conventional OpenGL textures are restricted to size dimensions that are powers of two.

The phrases POT (power of two) and NPOT (non-power of two) textures are used in the Overview and Issues section of this specification, but notice these terms are never required in the actual extension language to amend the core specification.

2. Should any enable or other state change be required to relax the texture dimension restrictions?

STATUS: RESOLVED

RESOLUTION: No. The restrictions on texture dimensions in the core OpenGL specification are enforced by errors. Extensions are free to make legal and defined the error behavior of extensions. This extension is really no different in that respect.

The argument for having an enable to "unlock" more generalized texture dimensions is that it avoids developers accidently releasing applications developed on an OpenGL implementation supporting this extension and unintentionally using NPOT textures. This situation exists in theory with other extensions that do not require new entry points or enumerants to operate (think of NV_blend_square). The real responsibility falls on developers to not use extensions unless the implementation advertises support for the extension and do proper testing to ensure this is really the case.

An additional issue with not having an enable to "unlock" this feature concerns the cases where existing apps might actually be relying on the current error condition to tell them what to do, but might not be able to handle the "new" success this extension would create. However, this seems to be limited to apps that are explicitly checking for implementation correctness (like a conformance test) and this does not seem to be a typical problem for "real-world" applications. The working group members agreed that it is acceptable to require those few apps which fall into this category to be updated in the context of this extension.

3. Should this extension be limited to a subset of conventional texture targets?

STATUS: RESOLVED

SUGGESTION: No. This extension should apply to 1D, 2D, 3D, and cube map textures (all supported by OpenGL 1.4) but this extension does NOT extend or otherwise affect the EXT_texture_rectangle extension's TEXTURE_RECTANGLE_EXT target.

```
  One early point of debate was whether we should have a single
  unified extension which lifted the power of two restrictions from
  all targets, or whether we should have individual target specific
  extensions.   For example, one could imagine separate extensions for
  ARB_texture_non_power_of_two_2d, ARB_texture_non_power_of_two_3d,
  ARB_texture_non_power_of_two_cube_map.

  The advantages of the separate extension approach are to allow IHV's
  to choose which pieces of functionality to support independently.
  The advantages of the single extension approach is to have a
  simpler and more forward looking extension.
```

4. Are cube map texture images still required to be square when this extension is supported?

```
  STATUS: RESOLVED

  RESOLUTION:  Yes.  But while the width and height of each level
  must be equal, they can be NPOT.
```

5. How is a conventional NPOT target different from the texture rectangle target?

```
  STATUS: RESOLVED

  RESOLUTION:
  The biggest practical difference is that coventional targets use
  normalized texture coordinates (ie, [0..1]) while the texture
  rectangle target uses unnormalized (ie, [0..w]x[0..h]) texture
  coordinates.

  Differences include:

  + In ARB_texture_non_power_of_two:
    * mipmapping is allowed, default filter remains unchanged.
    * all wrap modes are allowed, default wrap mode remains unchanged.
    * borders are supported.
    * paletted textures are not unsupported.
    * texture coordinates are addressed parametrically [0..1],[0..1]
  + In EXT_texture_rectangle:
    * mipmapping is not allowed, default filter is changed to LINEAR.
    * only CLAMP* wrap modes are allowed, default is CLAMP_TO_EDGE.
    * borders are not supported.
    * paletted textures are unsupported.
    * texture coordinates are addressed non-parametrically [0..w],[0..h].
```

6. What is the dimension reduction rule for each successively smaller mipmap level?

STATUS: RESOLVED

RESOLUTION: Each successively smaller mipmap level is half the size of the previous level, but if this half value is a fractional value, you should round down to the next largest integer. Essentially:

```
max(1, floor(w_b / 2^i)) x
   max(1, floor(h_b / 2^i)) x
      max(1, floor(d_b / 2^i))
```

where i is the ith level beyond the 0th level (the base level).

This is a "floor" convention. An alternative is a "ceiling" convention.

The primary reason to favor the floor convention is that Direct3D uses the floor convention.

Also, the "ceiling" convention potentially implies one more mipmap level than the "floor" convention.

Some regard the "ceiling" convention to have nicer properties with respect to making sure that each level samples at at least 2x the frequency of the next level. This can reduce the chances of sampling artifacts. However, it's probably not worth diverging from the Direct3D convention just for this. A more sophisticated downsampling algorithm (using a larger kernel perhaps) during mipmap level generation can help reduce artifacts related to using the "floor" convention.

The "floor" convention has a relatively straightforward way to evaluate (with integer math) means to determine how many mipmap levels are required for a complete pyramid:

```
numLevels = 1 + floor(log2(max(w, h, d)))
```

The "floor" convention can be evaluated incrementally with the following recursion:

```
 nextLODdim = max(1, currentLODdim >> 1)
```

where currentLODdim is the dimension of a level N and nextLODdim is the dimension of level N+1. The recursion stops when level numLevels-1 is reached.

Other compromise rules exist such as "round" (floor(x+0.5)). Such a hybrid approach make it more difficult to compute how many mipmap levels are required for a complete pyramid.

Note that this extension is compatible with supporting other rules because it merely relaxes the error and completeness conditions for mipmaps. At the same time, it makes sense to provide developers a single consistent rule since developers are unlikely to want to generate mipmaps for different rules unnecessarily. One reasonable rule is sufficient and preferable, and the "floor" convention is the best choice.

7. Should the LOD for filtering (rho) be computed differently for NPOT textures?

STATUS: RESOLVED

RESOLUTION: No (though, ideally, the answer would be "yes slightly somehow"). The core OpenGL specification already allows that the ideal computation of rho (even for POT textures) is "often impractical to implement". The "ceiling" convention adds one more mipmap level for NPOT textures so at extreme minification, the "ceiling" convention may be somewhat sharper than ideal (whereas "floor" would be blurrier).

This excess bluriness should only be significant at the smallest (blurriest) mipmap levels where it should be quite difficult to notice for properly downsampled mipmap images.

8. Should there be any restrictions on the wrap modes supported for NPOT textures?

STATUS: RESOLVED

RESOLUTION: No restrictions; all existing wrap modes (GL_REPEAT, GL_CLAMP, GL_CLAMP_TO_EDGE, GL_CLAMP_TO_BORDER, and GL_MIRRORED_REPEAT) should "just work" with NPOT textures.

The difficult part of this requirement is to compute "mod w_i" (or h_i or d_i) rather than simply "mod 2^n" (or 2^m or 2^l) for the GL_REPEAT wrap mode (GL_MIRRORED_REPEAT may also be an issue, but as defined by OpenGL 1.4, no "mod" math is required to implement the mirrored repeat wrap mode). REPEAT is too commonly used (indeed it is the default wrap mode) to exclude it for NPOT textures.

9. How does this extension interact with ARB_texture_compression?

STATUS: RESOLVED

RESOLUTION: It does not. ARB_texture_compression doesn't technically require that any compressed formats be supported. Implementations can choose to compress or not compress any particular texture.

While implementations may choose an internal component resolution and compressed format, the OpenGL 1.4 requires that the choice be a function only of the TexImage parameters. If an implementation chose not to compress NPOT textures, it might get into a situation where a 7x7 image wasn't compressed but its 4x4, 2x2, and 1x1 mipmaps were compressed. The result would be an inconsistent mipmap chain since the internal format of each level would not the same.

Therefore, an implementation must be able to handle the case where decisions it makes during image specification can be corrected appropriately at render time. This may mean that an implementation such as the one described above may need to tempoarily keep compressed and uncompressed images internally until the full mipmap stack can be examined or may need to decompress previously compressed images in order to recover.

10. How does this extension interact with specific texture compression extensions such as EXT_texture_compression_s3tc?

STATUS: RESOLVED

RESOLUTION: It does not. If both this extension and EXT_texture_compression_s3tc are supported, applications can safely load NPOT S3TC-compressed textures.

Textures are still decomposed into an array of 4x4 blocks. The compressed data for any texels outside the specified image dimensions are irrelevant and are effectively ignored, just as they are for the 1x1 and 2x2 mipmaps of a POT S3TC-compressed texture.

11. How is automatic mipmap generation affected by this extension?

STATUS: RESOLVED

RESOLUTION: It is not directly affected. If an implementation supports automatic mipmap generation, then mipmap generation must be supported even for NPOT textures.

Note however, that the OpenGL 1.4 specification recommends a "2x2 box filter" for the default filter. This is typo since a 2x2 box filter would be incorrect for 1D and 3D textures. With support for NPOT textures, this "2x2 box filter" becomes even more inappropriate. The wording should be changed to simply recommend a box filter where the dimensionality and filter size is assumed appropriate for the texture image dimensionality and size.

12. Are any edits required for Section 3.8.10 "Texture Completeness"?

STATUS: RESOLVED

RESOLUTION: No. This section references Section 3.8.8 for the allowed sequence of dimensions for completeness (rather than stating the requirements explicition in Section 3.8.10). The only difference between NPOT and POT textures is the allowable sequence of mipmap sizes, and in both cases, a smaller level is half the size of the larger (modulo rounding).

As with POT textures, a mipmap chain is consistent only if the correct sequence of sizes is found. As with POT textures, an attempt to load a mipmap that could never be part of a consistent mipmap chain should fail. For example, if an implementation supports textures with dimensions only up to 1024, an attempt to load level 2 with a 257x114 texture will fail because the smallest possible corresponding level 0 texture would have to be 1028x456.

13. The WGL_ARB_render_texture extension allows creating a pbuffer with the WGL_PBUFFER_LARGEST_ARB attribute. If this extension is present, should this attribute potentially return a NPOT pbuffer?

STATUS: UNRESOLVED

SUGGESTION: The WGL_ARB_render_texture specification appears to anticipate NPOT textures with this statement: "e.g. Both the width and height will be a power of 2 if the implementation only supports power of 2 textures." so I think the right thing to do is allow NPOT textures (of the proper aspect ratio) to be returned.

It is not entirely clear if this behavior is "safe" for preexisting applications that might not be aware of NPOT textures. The safe thing would be to add a WGL_PBUFFER_LARGEST_NPOT_ARB enumerant that could return NPOT textures and require that the existing WGL_PBUFFER_LARGEST_ARB enumerant always return POT textures.

New Procedures and Functions

None

New Tokens

None

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

None

Additions to Chapter 3 of the GL Specification (Rasterization)

-- Section 3.8.1 "Texture Image Specification"

Replace the discussion of the border parameter with:

"The border argument to TexImage3D is a border width. The significance of borders is described below. The border width affect the dimensions of the texture image; it must be the case that

$$w_s = w_i + 2 b_s \qquad (3.13)$$

$$h_s = h_i + 2 b_s \qquad (3.14)$$

$$d_s = d_i + 2 b_s \qquad (3.15)$$

where w_s, h_s, and d_s are the specified image width, height, and depth, and w_i, h_i, and d_i are the dimensions of the texture image internal to the border. If w_i, h_i, or d_i are less than zero, then the error INVALID_VALUE is generated.

-- Section 3.8.8 "Texture Minification"

In the subsection "Scale Factor and Level of Detail"...

Replace the sentence defining the u, v, and w functions with:

"Let u(x,y) = w_i * s(x,y), v(x,y) = h_i * t(x,y), and w(x,y) = d_i * r(x,y), where w_i, h_i, and d_i are as defined by equations 3.13, 3.14, and 3.15 with w_s, w_s, and d_s equal to the width, height, and depth of the image array whose level is TEXTURE_BASE_LEVEL."

Replace 2^n, 2^m, and 2^l with w_i, h_i, and d_i in Equations 3.19, 3.20, and 3.21.

```
    { floor(u),   s < 1
i = {                               (3.19)
    { w_i - 1,    s = 1

    { floor(u),   t < 1
j = {                               (3.20)
    { h_i - 1,    t = 1

    { floor(u),   r < 1
k = {                               (3.21)
    { d_i - 1,    r = 1
```

Replace 2^n, 2^m, and 2^l with w_i, h_i, and d_i in the equations for computing i_0, j_0, k_0, i_1, j_1, and k_1 used for LINEAR filtering.

```
      { floor(u - 1/2) mod w_i,   TEXTURE_WRAP_S is REPEAT
i_0 = {
      { floor(u - 1/2),           otherwise

      { floor(v - 1/2) mod h_i,   TEXTURE_WRAP_T is REPEAT
j_0 = {
      { floor(v - 1/2),           otherwise

      { floor(w - 1/2) mod d_i,   TEXTURE_WRAP_R is REPEAT
k_0 = {
      { floor(w - 1/2),           otherwise

      { (i_0 + 1) mod w_i,        TEXTURE_WRAP_S is REPEAT
i_1 = {
      { i_0 + 1,                  otherwise

      { (j_0 + 1) mod h_i,        TEXTURE_WRAP_T is REPEAT
j_1 = {
      { j_0 + 1,                  otherwise

      { (k_0 + 1) mod d_i,        TEXTURE_WRAP_R is REPEAT
k_1 = {
      { k_0 + 1,                  otherwise
```

In the subsection "Mipmapping"...

Replace the last sentence of the first paragraph with:

"If the image array of level level_base, excluding its border, has dimensions w_b x h_b x d_b, then there are floor(log2(max(w_b, h_b, d_b))) + 1 image arrays in the mipmap. Numbering the levels such that level level_base is the 0th level, the ith array has dimensions

```
  max(1, floor(w_b / 2^i)) x
    max(1, floor(h_b / 2^i)) x
      max(1, floor(d_b / 2^i))
```

until the last array is reached with dimension 1 x 1 x 1."

Replace the second sentence of the second paragraph with:

"Level-of-detail numbers proceed from level_base for the original texture array through p = floor(log2(max(w_b, h_b, d_b))) + level_base with each unit increase indicating an array of half the dimensions of the previous one (rounded down to the next integer if fractional) as already described."

In the subsection "Automatic Mipmap Generation"...

Replace the second sentence of the third paragraph with:

"No particular filter algorithm is required, though a box filter is recommended as the default filter."

-- Section 3.8.10 "Texture Completeness"

In the subsection "Effects of Completeness on Texture Image Specification"...

Replace the last sentence with:

"A mipmap complete set of arrays is equivalent to a complete set of arrays where level_base = 0 and level_max = 1000, and where, excluding borders, the dimensions of the image array being created are understood to be half the corresponding dimensions of the next lower numbered array (rounded down to the next integer if fractional)."

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the GL Specification (Special Functions)

None

Additions to the GLX Specification

None

Additions to the EXT_texture_compression_s3tc and NV_texture_compression_vtc Specification

```
    Add this paragraph:

    "For a compressed texture where w_i != 2^m OR h_i != 2^n OR d_i != 2^l
    for some integer value of m, n, and l, the 4x4 tiles are assumed to be
    aligned to u=0, v=0, w=0 origin in texel space.  For such compressed
    textures, this implies that texels in regions of tiles beyond the
    edges u=w_i, v=h_i, and w=d_i will not be sampled explicitly."
```

GLX Protocol

```
    None
```

Errors

```
    Various errors are ELIMINATED when this extension is supported as
    noted.

    INVALID_VALUE is NO LONGER generated by TexImage1D or glCopyTexImage1D
    if width is not zero or cannot be represented as 2^n+2(border)
    for some integer value of n.

    INVALID_VALUE is NO LONGER generated by TexImage2D or glCopyTexImage2D
    if width or height is not zero or cannot be represented as
    2^n+2(border) for some integer value of n.

    INVALID_VALUE is NO LONGER generated by TexImage3D if width, height,
    or depth is not zero or cannot be represented as 2^n+2(border)
    for some integer value of n.
```

New State

```
    None
```

New Implementation Dependent State

```
    None
```

Revision History

```
    Date 05/14/2004
    Revision: 1.0
       -  Formated text for 72 column convention
       -  Fixed date for last revision
       -  fix "Image2d" typo

    Date: 03/23/2004
    Revision: 1.0
        - Formulas for computing the dimensions of mipmap sizes based
          on the base level size should involve 2^i (not i^2)
    Date: 09/11/2003
    Revision: 1.0
        - allow zero (instead of just positive values before) when
          specifying the width, height, and depth of texture image
          dimensions; this is to avoid an inconsistency with the
          sample implementation

    Date: 05/29/2003
    Revision: 0.10
        - removed "@" language for target specific behavior, the spec
          now treats all targets uniformly

    Date: 05/21/2003
    Revision: 0.9
        - fixed typo: ARB/IBM_mirrored_repeat should have been
          ARB/IBM_texture_mirrored_repeat
        - fixed various other minor typos, duplicated words, etc.
        - added a line to issue #6 regarding suggesting use of a
          larger kernel when downsampling using the floor convention
        - coalesced the equations that used 3 2-term max equations into
          single 3-term max equations for clarity
        - fixed two more typos where "ceil" should have been "floor"
        - refer to ARB_texture_rectangle as EXT_texture_rectangle
          (this may change back when/if back extension becomes ARB'ified)

    Date: 05/10/2003
    Revision: 0.8
        - additional additional names to contributors list
        - clarified language describing resolution of issues #9,10,11

    Date: 05/08/2003
    Revision: 0.7
        - very minor language update to overview section regarding
          exporting of ARB_texture_non_power_of_two string
        - fixed another two places where it said we should round up
          instead of down (in section 3.8.10 "Texture Completeness",
          and in section 3.8.8 "Texture Minification")
        - mark the regions of the spec affected by the decision to
          use separate strings per texture target with the "@" symbol.
          This is temporary until issue #3 is resolved.
        - resolved issues 9,10,11,12

    Date: 05/08/2003
    Revision: 0.6
        - updated revision history and coalesced revision notes from
          various specs
        - fixed typo in issue #5 ("2d" --> "non_power_of_two")
        - clarified the discussion in issue #3 as the langage was a
          little confusing in parts.
        - explicitly refer to the cube map targets in section 3.8.1
          instead of using the "made up" target TEXTURE_CUBE_MAP.
    Date: 05/06/2003
    Revision: 0.5
        - changed name of extension from ARB_texture_np2 to
          ARB_texture_non_power_of_two
        - added target specific extension strings
        - added more discussion to several issues based on feedback from
          the working group meetings
        - fixed several typos where INVALID_VALUE was INVALID_VALID
        - addressed typo in issue #6, it said you should round up,
          but really we agreed to round down when describing the mipmap
          stack (floor vs ceil convention).
        - resolved issues 1 - 8.

    Date: 04/24/2003
    Revision: 0.4 (jsandmel)
        - numbered issues list
        - additional discussion of several issues
        - added more explicit comparison of texture_rectangle and this
          proposal

    Date: 04/10/2003
    Revision: 0.3 (mjk)
        - integrates input from the ARB_texture_2d_np2 proposals.

    Date: 03/25/2003
    Revision: 0.1 (jsandmel)
        - draft proposal
        - deals with 2d targets only
        - named: ARB_texture_2d_np2
```

Name

    ARB_texture_rectangle

Name Strings

    GL_ARB_texture_rectangle

Contributors

    Pat Brown
    Daniel Ginsburg
    Michael Gold
    Mark J. Kilgard
    Jon Leech
    Bill Licea-Kane
    Barthold Lichtenbelt
    Benjamin Lipchak
    Brian Paul
    John Rosasco
    Jeremy Sandmel
    Geoff Stahl

Contact

    Mark J. Kilgard, NVIDIA Corporation (mjk 'at' nvidia.com)
    Geoff Stahl, Apple Computer (gstahl 'at' apple.com)

Notice

Status

    Complete. Approved by the ARB on June 8, 2004.

    Amended language re-voted by the ARB on November 3, 2005.

    Functionally identical to EXT_texture_rectangle and
    NV_texture_rectangle extensions currently shipping, except for
    the additions to the OpenGL Shading Language.

Version

    Date: October 4, 2005
    Revision: 1.21

Number

    ARB Extension #38

Dependencies

    OpenGL 1.1 is required

    OpenGL 1.4 (or ARB_texture_mirrored_repeat) affects the definition
    of this extension.

ARB_texture_non_power_of_two trivially affects the definition of this extension.

ATI_texture_mirror_once affects the definition of this extension.

EXT_paletted_texture affects the definition of this extension.

EXT_texture_compression_s3tc affects the definition of this extension.

EXT_texture_mirror_clamp affects the definition of this extension.

The OpenGL Shading Language specification (provided by OpenGL 2.0 and/or ARB_shader_objects) interacts with this extension.

This extension is written against the OpenGL 2.0 specification.

Overview

OpenGL texturing is limited to images with power-of-two dimensions and an optional 1-texel border. The ARB_texture_rectangle extension adds a new texture target that supports 2D textures without requiring power-of-two dimensions.

Non-power-of-two sized (NPOTS) textures are useful for storing video images that do not have power-of-two sized (POTS). Re-sampling artifacts are avoided and less texture memory may be required by using non-power-of-two sized textures. Non-power-of-two sized textures are also useful for shadow maps and window-space texturing.

However, non-power-of-two sized textures have limitations that do not apply to power-of-two sized textures. NPOTS textures may not use mipmap filtering; POTS textures support both mipmapped and non-mipmapped filtering. NPOTS textures support only the GL_CLAMP, GL_CLAMP_TO_EDGE, and GL_CLAMP_TO_BORDER wrap modes; POTS textures support GL_CLAMP_TO_EDGE, GL_REPEAT, GL_CLAMP, GL_MIRRORED_REPEAT, and GL_CLAMP_TO_BORDER (and GL_MIRROR_CLAMP_ATI and GL_MIRROR_CLAMP_TO_EDGE_ATI if ATI_texture_mirror_once is supported) . NPOTS textures do not support an optional 1-texel border; POTS textures do support an optional 1-texel border.

NPOTS textures are accessed by dimension-dependent (aka non-normalized) texture coordinates. So instead of thinking of the texture image lying in a [0..1]x[0..1] range, the NPOTS texture image lies in a [0..w]x[0..h] range.

This extension adds a new texture target and related state (proxy, binding, max texture size).

Issues

1) *Should rectangular textures simply be an extension to the 2D texture target that allows non-power-of-two widths and heights?*

No. The rectangular texture is an entirely new texture target type called GL_TEXTURE_RECTANGLE_ARB. This is because while the texture rectangle target relaxes the power-of-two dimensions requirements of the texture 2D target, it also has limitations such as the absence of both mipmapping and the GL_REPEAT and GL_MIRRORED_REPEAT wrap modes. Additionally, rectangular textures do not use [0..1] normalized texture coordinates.

The texture rectangle is an analogue to the pixel rectangle primitive (see section 3.6 titled "Pixel Rectangles" in the core specification) and the framebuffer. Just as the pixel rectangle primitive and the framebuffer are accessed by integer-ized dimension-dependent 2D coordinates, so is the texture rectangle. Just as pixel rectangles and the framebuffer do not have mipmaps, nor do texture rectangles.

2) *Should 1D, 2D, 3D, or cube map textures be allowed to be NPOTS by this extension?*

No. The ARB_texture_non_power_of_two extension relaxes the power-of-two restrictions for these conventional texture targets to support NPOTS while maintaining the normalized texture coordinates.

3) *How is the image of a rectangular texture specified?*

Using the standard OpenGL API for specifying a 2D texture image: glTexImage2D, glSubTexImage2D, glCopyTexImage2D, and glCopySubTexImage2D. The target for these commands is GL_TEXTURE_RECTANGLE_ARB though.

This is similar to how the texture cube map functionality uses the 2D texture image specification API though with its own texture target.

The texture target GL_TEXTURE_RECTANGLE_ARB should also be used for glGetTexImage, glGetTexLevelParameteriv, and glGetTexLevelParameterfv.

4) *Should anything be said about performance?*

No, but developers should not be surprised if conventional POTS textures will render slightly faster than texture rectangle textures. This is particularly likely to be true when texture rectangle textures are minified leading to texture cache thrashing due to lack of support for mipmaps.

5) *Is mipmap filtering permitted?*

Mipmap filtering is not permitted. Since this is the case the default minification filter for GL_TEXTURE_RECTANGLE_ARB targets is GL_LINEAR.

6) *What texture wrap modes are allowed and what is the default state?*

Only the GL_CLAMP, GL_CLAMP_TO_EDGE, and CLAMP_TO_BORDER wrap modes are allowed. CLAMP_TO_EDGE is the default state. GL_REPEAT and GL_MIRRORED_REPEAT are not supported with the GL_TEXTURE_RECTANGLE_ARB texture target.

```
 7) Are texture borders supported?

    Borders are not supported.

 8) Are paletted textures supported?

    Paletted rectangular textures are not supported.

 9) Can compressed texture images be specified for a rectangular texture?

    The generic texture compression internal formats introduced by
    ARB_texture_compression are supported for rectangular textures
    because the image is not presented as compressed data and the
    ARB_texture_compression extension always permits generic texture
    compression internal formats to be stored in uncompressed form.
    Implementations are free to support generic compression internal
    formats for rectangular textures if supported but such support is
    not required.

    This extensions makes a blanket statement that specific compressed
    internal formats for use with glCompressedTexImage<n>D are NOT
    supported for rectangular textures.  This is because several existing
    hardware implementations of texture compression formats such as S3TC
    are not designed for compressing rectangular textures.  This does
    not preclude future texture compression extensions from supporting
    compressed internal formats that do work with rectangular extensions
    (by relaxing the current blanket error condition).

10) How are rectangular textures enabled?

    Rectangular textures are enabled by enabling the
    GL_TEXTURE_RECTANGLE_ARB texture target via glEnable
    (GL_TEXTURE_RECTANGLE_ARB). This enable is prioritized above
    GL_TEXTURE_2D and below GL_TEXTURE_3D.

    From lowest priority to highest priority: GL_TEXTURE_1D,
    GL_TEXTURE_2D, GL_TEXTURE_RECTANGLE_ARB, GL_TEXTURE_3D,
    GL_TEXTURE_CUBE_MAP.

11) How are texture coordinates addressed for rectangular textures?

    Texture coordinates are addressed without being normalized from
    [0..1], instead [0..w] and [0..h] are used, where w and h are width
    and height of the texture respectively.

12) How should applications determine the available maximum texture
    dimensions available?

    Implementation dependent rectangular texture size limitations are
    queried using the GL_MAX_RECTANGLE_TEXTURE_SIZE_ARB parameter and
    may be different that standard texture size limits.

13) How does the handling of the R texture component differ from
    the handling of S and T?

    The R texture coordinate for rectangular textures is handled
    as it would be for standard two dimensional textures.  Thus the
```

coordinates range from [0..1] and the wrapping mode is unchanged from the default.

14) *Does this extension work with OpenGL 1.4's shadow mapping?*

Yes. The one non-obvious allowance to support OpenGL 1.4's shadow mapping is that the R texture coordinate wrap mode remains UNCHANGED for rectangular textures. Clamping of the R texture coordinate for rectangular textures uses the standard [0,1] interval rather than the [0,w_s] or [0,h_s] intervals as in the case of S and T. This is because R represents a depth value in the [0,1] range whether using a 2D or rectangular texture.

15) *How does this extension interact with GLSL based on the "OpenGL Shading Language Extension Conventions"?*

Unfortunately, this extension was specified and implemented contemporaneously with the GLSL Extension Conventions and because of this timing does not follow its guidance for #extension and adornment of new GLSL names. Because this extension has both an API interaction (adding a new rectangle texture target) and a GLSL interaction (functions and sampler types for accessing texture rectangles), you can't practically use the GLSL texture rectangle functionality without the API functionality. For this reason, detecting the GL_ARB_texture_rectangle string is sufficient for assuming the GLSL functionality is present.

Conceptually, you can consider the declaration #extension GL_ARB_texture_rectangle : require, to allow support for texture rectangles, to be implicitly prepended to every GLSL shader when ARB_texture_rectangle is advertised.

All future GLSL extensions should follow the "OpenGL Shading Language Extension Conventions" however.

16) *How can a GLSL shader tell if this extension is supported?*

"GL_ARB_texture_rectangle" preprocessor macro is predefined to be 1.

17) *Should GL_SAMPLER_2D_RECT_ARB and GL_SAMPLER_2D_RECT_SHADOW_ARB be returned by the "type" parameter of glGetActiveUniformARB when returning the type of a sampler2DRect or sampler2DRectShadow sampler uniform?*

Yes, there is already language in the ARB_shader_objects extension saying this so there's no additional language added to this extension. The language is missing from OpenGL 2.0 so we add the ARB_shader_objects language as part of this specification too.

18) *Can a shader still turn off support for this extension?*

Yes, a shader can still include all variations of #extension GL_ARB_texture_rectangle in its source code. This includes #extension GL_ARB_texture_rectangle : disable, to disable support for it.

New Procedures and Functions

None

New Tokens

Accepted by the <cap> parameter of Enable, Disable and IsEnabled; by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv and GetDoublev; and by the <target> parameter of BindTexture, GetTexParameterfv, GetTexParameteriv, TexParameterf, TexParameteri, TexParameterfv and TexParameteriv:

```
TEXTURE_RECTANGLE_ARB            0x84F5
```

Accepted by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv and GetDoublev:

```
TEXTURE_BINDING_RECTANGLE_ARB    0x84F6
```

Accepted by the <target> parameter of GetTexLevelParameteriv, GetTexLevelParameterfv, GetTexParameteriv and TexImage2D:

```
PROXY_TEXTURE_RECTANGLE_ARB      0x84F7
```

Accepted by the <pname> parameter of GetBooleanv, GetDoublev, GetIntegerv and GetFloatv:

```
MAX_RECTANGLE_TEXTURE_SIZE_ARB   0x84F8
```

Accepted by the <target> parameter of GetTexImage, GetTexLevelParameteriv, GetTexLevelParameterfv, TexImage2D, CopyTexImage2D, TexSubImage2D and CopySubTexImage2D:

```
TEXTURE_RECTANGLE_ARB
```

Returned by <type> parameter of GetActiveUniform when the location <index> for program object <program> is of type sampler2DRect:

```
SAMPLER_2D_RECT_ARB              0x8B63
```

Returned by <type> parameter of GetActiveUniform when the location <index> for program object <program> is of type sampler2DRectShadow:

```
SAMPLER_2D_RECT_SHADOW_ARB       0x8B64
```

Additions to Chapter 2 of the OpenGL 2.0 Specification (OpenGL Operation)

- (2.15.3, pg. 80-81) "Uniform Variables" under "Shader Variables"

Add SAMPLER_2D_RECT_ARB and SAMPLER_2D_RECT_SHADOW_ARB to the list of returned types in the sentence starting "The type returned can be any of ..."

- (2.15.4, pg. 86) "Texture Access" under "Shader Execution"

Replace the three bullets with the following language:

"...the results of a texture lookup are undefined if:

* The sampler used in a texture lookup function is of type sampler1D or sampler2D or sampler2DRect, and the texture object's internal format is DEPTH_COMPONENT, and the TEXTURE_COMPARE_MODE is not NONE.

* The sampler used in a texture lookup function is of type sampler1DShadow or sampler2DShadow or sampler2DRectShadow, and the texture object's internal format is DEPTH_COMPONENT, and the TEXTURE_COMPARE_MODE is NONE.

* The sampler used in a texture lookup function is of type sampler1DShadow or sampler2DShadow or sampler2DRectShadow, and the texture object's internal format is not DEPTH_COMPONENT."

Additions to Chapter 3 of the OpenGL 2.0 Specification (Rasterization)

These changes describe use of the TEXTURE_RECTANGLE_ARB texture target, supported formats, texture dimensions, and texture proxies:

- (3.6.3, pg. 118) "Pixel Transfer Modes" under "Color Table Specification" or the ColorTableEXT description in the EXT_paletted_texture specification

If EXT_paletted_texture is supported, add the following statement after paragraph 5 of the sub-section:

"The error INVALID_ENUM is generated if the target to ColorTable (or ColorTableEXT or the various ColorTable and ColorTableEXT alternative commands) is TEXTURE_RECTANGLE_ARB or PROXY_TEXTURE_RECTANGLE_ARB."

- (3.8.1, p. 151) "Texture Image Specification"

Change the first sentence of the fourth paragraph on this page to:

Textures with a base internal format of DEPTH COMPONENT are supported by texture image specification commands only if target is TEXTURE_1D, TEXTURE_2D, TEXTURE_RECTANGLE_ARB, PROXY_TEXTURE_1D, PROXY_TEXTURE_2D or PROXY_TEXTURE_RECTANGLE_ARB.

- (3.8.1, pg. 156) "Texture Image Specification"

Add a sentence to the middle of the 20th paragraph of the section (first paragraph on the page), directly after "... for image arrays of level 0 through k, where k is the log base 2 of MAX_TEXTURE_SIZE." reading:

"The maximum allowable width of a rectangular texture image, and the maximum allowable height of a rectangular texture image, must be at least the implementation-dependent value of MAX_RECTANGLE_TEXTURE_SIZE_ARB."

- (3.8.1, pg. 156) "Texture Image Specification"

In the 22th paragraph of this section (sixth paragraph on the page), change the sentence following "The command void TexImage2D ... a two-dimensional texture image." through the rest of the paragraph in the section describing two-dimensional texturing to read:

"<target> must be one of TEXTURE_2D for a two-dimensional texture, or one of TEXTURE_RECTANGLE_ARB for a rectangle texture, or one of TEXTURE_CUBE_MAP_POSITIVE_X, TEXTURE_CUBE_MAP_NEGATIVE_X, TEXTURE_CUBE_MAP_POSITIVE_Y, TEXTURE_CUBE_MAP_NEGATIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z, or TEXTURE_CUBE_MAP_NEGATIVE_Z for a cube map texture. Additionally, <target> may be either PROXY_TEXTURE_2D for a two-dimensional proxy texture, PROXY_TEXTURE_RECTANGLE_ARB for a rectangle proxy texture or PROXY_TEXTURE_CUBE_MAP for a cube map proxy texture as discussed in section 3.8.10. The other parameters match the corresponding parameters of TexImage3D."

Add this paragraph following the above two-dimensional texturing introduction, reading:

When the target is TEXTURE_RECTANGLE_ARB, the INVALID_VALUE error is generated if border is any value other than zero or the level is any value other than zero. In the case of a rectangular texture, ws and hs equal the specified width and height respectively of the rectangular texture image while ds is 1."

If EXT_paletted_texture is supported, add this paragraph too:

"Rectangular textures do not support paletted formats. The error INVALID_ENUM is generated if the target is TEXTURE_RECTANGLE_ARB or PROXY_TEXTURE_RECTANGLE_ARB and the format is COLOR_INDEX or the internal format is COLOR_INDEX or one of the COLOR_INDEX<n>_EXT internal formats."

- (3.8.1, pg. 156) "Texture Image Specification"

Amend the fourth paragraph on the page to read:

"A two-dimensional texture consists of a single two-dimensional texture image. A rectangle texture consists of a single 2D texture image. A cube map texture is a set of six two-dimensional texture images. The six cube map texture targets form a single cube map texture though each target names a distinct face of the cube map. The TEXTURE_CUBE_MAP_* targets listed above update their appropriate cube map face 2D texture image. The six cube map two-dimensional image tokens such as TEXTURE_CUBE_MAP_POSITIVE_X are used when specifying, updating, or querying one of a cube map's six two-dimensional images, but when enabling cube map texturing or binding to a cube map texture object (that is when the cube map is accessed as a whole as opposed to a particular two-dimensional image), the TEXTURE_CUBE_MAP target is specified."

- (3.8.1, pg. 157) "Texture Image Specification"

Append to the end of the third to the last paragraph in the section:

"A rectangular texture array has depth ds=1, with height hs and width ws defined by the specified image height and width parameters."

- (3.8.2, pg. 159) "Alternate Texture Image Specification Commands"

Add TEXTURE_RECTANGLE_ARB to the target list of the second paragraph of the section to say:

... "Currently, <target> must be TEXTURE_2D, TEXTURE_RECTANGLE_ARB, TEXTURE_CUBE_MAP_POSITIVE_X, TEXTURE_CUBE_MAP_NEGATIVE_X, TEXTURE_CUBE_MAP_POSITIVE_Y, TEXTURE_CUBE_MAP_NEGATIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z, or TEXTURE_CUBE_MAP_NEGATIVE_Z." ...

- (3.8.2, pg. 160) "Alternate Texture Image Specification Commands"

Add TEXTURE_RECTANGLE_ARB to the target list in the fifth paragraph of the section to say:

... "Currently the target arguments of TexSubImage1D and CopyTexSubImage1D must be TEXTURE_1D, the <target> arguments of TexSubImage2D and CopyTexSubImage2D must be one of TEXTURE_2D, TEXTURE_RECTANGLE_ARB, TEXTURE_CUBE_MAP_POSITIVE_X, TEXTURE_CUBE_MAP_NEGATIVE_X, TEXTURE_CUBE_MAP_POSITIVE_Y, TEXTURE_CUBE_MAP_NEGATIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z, or TEXTURE_CUBE_MAP_NEGATIVE_Z, and the <target> arguments of TexSubImage3D and CopyTexSubImage3D must be TEXTURE_3D." ...

Also append to the end of this paragraph:

"If target is TEXTURE_RECTANGLE_ARB and level is not zero, the error INVALID_VALUE is generated."

- (3.8.3, pg. 164) "Compressed Texture Images"

Add the following paragraph after the second paragraph in the section, which introduces the CompressedTexImage<n>D commands:

"The error INVALID_ENUM is generated if the target parameter to one of the CompressedTexImage<n>D commands is TEXTURE_RECTANGLE_ARB or PROXY_TEXTURE_RECTANGLE_ARB."

Add the following paragraph after introducing the CompressedTexSubImage<n>D commands:

"The error INVALID_ENUM is generated if the target parameter to one of the CompressedTexSubImage<n>D commands is TEXTURE_RECTANGLE_ARB or PROXY_TEXTURE_RECTANGLE_ARB."

- (3.8.4, pg. 166) "Texture Parameters"

Add TEXTURE_RECTANGLE_ARB to paragraph one to say:

... "<target> is the target, either TEXTURE_1D, TEXTURE_2D, TEXTURE_RECTANGLE_ARB, TEXTURE_3D, or TEXTURE_CUBE_MAP." ...

- (3.8.4, pg. 168) "Texture Parameters"

Add the following paragraph to the end of the section:

"Certain texture parameter values may not be specified for textures with a target of TEXTURE_RECTANGLE_ARB. The error INVALID_ENUM is generated if the target is TEXTURE_RECTANGLE_ARB and the TEXTURE_WRAP_S, TEXTURE_WRAP_T, or TEXTURE_WRAP_R parameter is set to REPEAT, MIRRORED_REPEAT, MIRROR_CLAMP_EXT (MIRROR_CLAMP_ATI), MIRROR_CLAMP_TO_EDGE_EXT (MIRROR_CLAMP_TO_EDGE_ATI) or MIRROR_CLAMP_TO_BORDER_EXT. The error INVALID_ENUM is generated if the target is TEXTURE_RECTANGLE_ARB and the TEXTURE_MIN_FILTER is set to a value other than NEAREST or LINEAR (no mipmap filtering is permitted). The error INVALID_ENUM is generated if the target is TEXTURE_RECTANGLE_ARB and TEXTURE_BASE_LEVEL is set to any value other than zero."

- (3.8.7, pg. 170) "Texture Wrap Modes"

Add this final additional paragraph:

"Texture coordinates are clamped differently for rectangular textures. The r texture coordinate is wrapped as described above. When the texture target is TEXTURE_RECTANGLE_ARB, the s and t coordinates are wrapped as follows: CLAMP causes the s coordinate to be clamped to the range [0, wt]. CLAMP causes the t coordinate to be clamped to the range [0, ht]. CLAMP_TO_EDGE causes the s coordinate to be clamped to the range [0.5, wt-0.5]. CLAMP_TO_EDGE causes the t coordinate to be clamped to the range [0.5, ht - 0.5]. CLAMP_TO_BORDER causes the s coordinate to be clamped to the range [-0.5, wt + 0.5]. CLAMP_TO_BORDER causes the t coordinate to be clamped to the range [-0.5, ht + 0.5]."

- (3.8.8, pg. 171) "Texture Minification"

Under the "Scale Factor and Level of Detail" sub-section, change the fourth paragraph in the subsection to read:

"Let s(x,y) be the function that associates an s texture coordinate with each set of window coordinates (x,y) that lie within a primitive; define t(x,y) and r(x,y) analogously. For non-rectangular textures, let u(x,y) = wt * s(x,y), v(x,y) = ht * t(x,y), and w(x,y) = dt * r(x,y), where wt, ht, and dt are as defined by equations 3.15, 3.16, and 3.17 with ws, hs, and ds equal to the width, height, and depth of the image array whose level is level_base. However, for rectangular textures let u(x, y) = s(x, y), v(x, y) = t(x, y), and w(x, y) = r(x, y)."

- (3.8.8, pg. 173) "Texture Minification"

```
  Update the last sentence in the first partial paragraph on the page
  to read:

  "Depending on whether the texture's target is rectangular or
  non-rectangular, this means the texel at location (i,j,k) becomes
  the texture value, with i given by

          / floor (u),    s < 1 and non-rectangular texture
          |
  i =     | wt - 1,      s == 1 and non-rectangular texture (3.19)
          |
          | floor(u)      s < wt and rectangular texture
          |
          \ wt-1         s >= wt and rectangular texture

  (Recall that if TEXTURE_WRAP_S is REPEAT, then 0 <= s < 1.)
  Similarly, j is found as

          / floor(v),     t < 1 and non-rectangular texture
          |
  j =     | ht - 1,      t == 1 and non-rectangular texture (3.20)
          |
          | floor(v)      t < ht and rectangular texture
          |
          \ ht-1         t >= ht and rectangular texture

  and k is found as

          / floor (w),    r < 1
  k =     |                                               (3.21)
          \ dt - 1,      r == 1"
```

- (3.8.8, pg. 171) "Texture Minification"

```
  Change the last sentence in the first partial paragraph on the page,
  directly after equation 3.21 to read:

  "For a two-dimensional or rectangular texture, k is irrelevant; the
  texel at location (i,j) becomes the texture value."
```

- (3.8.8, pg. 174) "Texture Minification"

```
  Change the sentence preceding equation 3.26:

  "For a two-dimensional or rectangular texture,"
```

- (3.8.8, pg. 175) "Mipmapping"

```
  Follow the paragraph on the page which ends with "...  must be
  defined, as discussed in section 3.8.10." with:

  "Rectangular textures do not support mipmapping (it is an error to
  specify a minification filter that requires mipmapping)."
```

- (3.8.11, pg. 178) "Texture State and Proxy State"

Change the first sentence of the first paragraph to say:

"The state necessary for texture can be divided into two categories. First, there are the ten sets of mipmap arrays (one each for the one-, two-, and three-dimensional texture targets, one for the rectangular texture target (though the rectangular texture target has only one mipmap level), and six for the cube map texture targets) and their number." ...

- (3.8.11, pg. 179) "Texture State and Proxy State"

Change the sixth and fifth to last sentences of the first paragraph to say:

"In the initial state, the value assigned to TEXTURE_MIN_FILTER is NEAREST_MIPMAP_LINEAR, except for rectangular textures where the initial value is LINEAR, and the value for TEXTURE_MAG_FILTER is LINEAR. s, t, and r warp modes are all set to REPEAT, except for rectangular textures where the initial value is CLAMP_TO_EDGE."

- (3.8.11, pg. 179) "Texture State and Proxy State"

Change the second paragraph of the section to say:

"In addition to the one-, two-, three-dimensional, rectangular, and the six cube map sets of image arrays, the partially instantiated one-, two-, and three-dimensional, rectangular, and one cube map sets of proxy image arrays are maintained." ...

- (3.8.11, pg. 179) "Texture State and Proxy State"

Change the third paragraph to:

"One- and two-dimensional and rectangular proxy arrays are operated on in the same way when TexImage1D is executed with target specified as PROXY_TEXTURE_1D, or TexImage2D is executed with target specified as PROXY_TEXTURE_2D or PROXY_TEXTURE_RECTANGLE_ARB."

- (3.8.11, pg. 180) "Texture State and Proxy State"

Change the second sentence of the fifth paragraph of the section to:

"Therefore PROXY_TEXTURE_1D, PROXY_TEXTURE_2D, PROXY_TEXTURE_RECTANGLE_ARB, PROXY_TEXTURE_3D, and PROXY_TEXTURE_CUBE_MAP cannot be used as textures, and their images must never be queried using GetTexImage." ...

- (3.8.12, pg. 156) "Texture Objects"

Change the first sentence of the first paragraph to say:

"In addition to the default textures TEXTURE_1D, TEXTURE_2D, TEXTURE_RECTANGLE_ARB, TEXTURE_3D, and TEXTURE_CUBE_MAP, named one-dimensional, two-dimensional, rectangular, and three-dimensional texture objects and cube map texture objects can be created and operated on." ...

- **(3.8.12, pg. 180) "Texture Objects"**

  Change the second paragraph in the section to say:

  "A texture object is created by binding an unused name to TEXTURE_1D, TEXTURE_2D, TEXTURE_RECTANGLE_ARB, TEXTURE_3D, or TEXTURE_CUBE_MAP." ... "If the new texture object is bound to TEXTURE_1D, TEXTURE_2D, TEXTURE_RECTANGLE_ARB, TEXTURE_3D, or TEXTURE_CUBE_MAP, it remains a one-dimensional, two-dimensional, rectangular, three-dimensional, or cube map texture until it is deleted."

- **(3.8.12, pg. 180) "Texture Objects"**

  Change the third paragraph to say:

  "BindTexture may also be used to bind an existing texture object to either TEXTURE_1D, TEXTURE_2D, TEXTURE_RECTANGLE_ARB, TEXTURE_3D, or TEXTURE_CUBE_MAP."

- **(3.8.12, pg. 180) "Texture Objects"**

  Change paragraph five of the section to say:

  "In the initial state, TEXTURE_1D, TEXTURE_2D, TEXTURE_RECTANGLE_ARB, TEXTURE_3D, and TEXTURE_CUBE_MAP have one-dimensional, two-dimensional, rectangular, three-dimensional, and cube map state vectors associated with them respectively." ... "The initial, one-dimensional, two-dimensional, rectangular, three-dimensional, and cube map texture is therefore operated upon, queried, and applied as TEXTURE_1D, TEXTURE_2D, TEXTURE_RECTANGLE_ARB, TEXTURE_3D, and TEXTURE_CUBE_MAP respectively while 0 is bound to the corresponding targets."

- **(3.8.12, pg. 181) "Texture Objects"**

  Change paragraph six of the section to say:

  ... "If a texture that is currently bound to one of the targets TEXTURE_1D, TEXTURE_2D, TEXTURE_RECTANGLE_ARB, TEXTURE_3D, or TEXTURE_CUBE_MAP is deleted, it is as though BindTexture has been executed with the same <target> and <texture> zero." ...

- **(3.8.15 pg. 189) "Texture Application"**

  Replace the beginning sentences of the first paragraph with:

  "Texturing is enabled or disabled using the generic Enable and Disable commands, respectively, with the symbolic constants TEXTURE_1D, TEXTURE_2D, TEXTURE_RECTANGLE_ARB, TEXTURE_3D, or TEXTURE_CUBE_MAP to enable the one-dimensional, two-dimensional, rectangular, three-dimensional, or cube map texturing respectively. If both two- and one-dimensional textures are enabled, the two-dimensional texture is used. If the rectangular and either of the two- or one-dimensional textures is enabled, the rectangular texture is used. If the three-dimensional and any of the rectangular, two-dimensional, or one-dimensional textures is enabled, the three-dimensional texture is used. If the cube map texture and any of the three-dimensional, rectangular, two-dimensional, or one-dimensional textures is enabled, then cube map texturing is used.

- (3.11.2, pg. 195) "Texture Access" under "Shader Execution"

Replace the three bullets with the following language:

"...the results of a texture lookup are undefined if:

* The sampler used in a texture lookup function is of type sampler1D or sampler2D or sampler2DRect, and the texture object's internal format is DEPTH_COMPONENT, and the TEXTURE_COMPARE_MODE is not NONE.

* The sampler used in a texture lookup function is of type sampler1DShadow or sampler2DShadow or sampler2DRectShadow, and the texture object's internal format is DEPTH_COMPONENT, and the TEXTURE_COMPARE_MODE is NONE.

* The sampler used in a texture lookup function is of type sampler1DShadow or sampler2DShadow or sampler2DRectShadow, and the texture object's internal format is not DEPTH_COMPONENT."

Additions to Chapter 4 of the OpenGL 2.0 Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the OpenGL 2.0 Specification (Special Functions)

- (5.4, pg. 242) "Display Lists"

In the third to last paragraph of the section, add PROXY_TEXTURE_RECTANGLE_ARB to the list of PROXY_* tokens.

Additions to Chapter 6 of the OpenGL 2.0 Specification (State and State Requests)

- (6.1.3, pg. 247) "Enumerated Queries"

Change the fourth paragraph to say:

"The GetTexParameter parameter <target> may be one of TEXTURE_1D, TEXTURE_2D, TEXTURE_RECTANGLE_ARB, TEXTURE_3D, or TEXTURE_CUBE_MAP, indicating the currently bound one-dimensional, two-dimensional, rectangular, three-dimensional, or cube map texture object. For GetTexLevelParameter, <target> may be one of TEXTURE_1D, TEXTURE_2D, TEXTURE_RECTANGLE_ARB, TEXTURE_3D, TEXTURE_CUBE_MAP_POSITIVE_X, TEXTURE_CUBE_MAP_NEGATIVE_X, TEXTURE_CUBE_MAP_POSITIVE_Y, TEXTURE_CUBE_MAP_NEGATIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z, TEXTURE_CUBE_MAP_NEGATIVE_Z, PROXY_TEXTURE_1D, PROXY_TEXTURE_2D, PROXY_TEXTURE_RECTANGLE_ARB, PROXY_TEXTURE_3D, or PROXY_TEXTURE_CUBE_MAP, indicating the one-dimensional texture object, two-dimensional texture object, rectangular texture object, three-dimensional texture object, or one of the six distinct 2D images making up the cube map texture object or one-dimensional, two-dimensional, rectangular, three-dimensional, or cube map proxy state vector. Note that TEXTURE_CUBE_MAP is not a valid <target> parameter for GetTexLevelParameter because it does not specify a particular cube map face."

- (6.1.4, pg. 248) "Texture Queries"

Change the first paragraph to read:

... "It is somewhat different from the other get commands; <tex> is a symbolic value indicating which texture (or texture face in the case of a cube map texture target name) is to be obtained. TEXTURE_1D indicates a one-dimensional texture, TEXTURE_2D indicates a two-dimensional texture, TEXTURE_RECTANGLE_ARB indicates a rectangular texture, TEXTURE_3D indicates a three-dimensional texture, and TEXTURE_CUBE_MAP_POSITIVE_X, TEXTURE_CUBE_MAP_NEGATIVE_X, TEXTURE_CUBE_MAP_POSITIVE_Y, TEXTURE_CUBE_MAP_NEGATIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z, and TEXTURE_CUBE_MAP_NEGATIVE_Z indicate the respective face of a cube map texture."

- (6.1.4, pg. 249) "Texture Queries"

Add a final sentence to the fourth paragraph of the section, immediately after ... "or DEPTH COMPONENT causes the error INVALID ENUM.":

"Calling GetTexImage with a lod not zero when the tex is TEXTURE_RECTANGLE_ARB causes the error INVALID_VALUE."

Additions to version 1.10.59 of the OpenGL Shading Language specification

A new preprocessor #define is added to the OpenGL Shading Language:

```
    #define GL_ARB_texture_rectangle 1
```

Change the second to last paragraph on page 12 (#extension directive):

The initial state of the compiler is as if the directive

```
    #extension all : disable
```

was issued, telling the compiler that all error and warning reporting must be done according to this specification, ignoring any extensions. The only execption to this rule is the GL_ARB_texture_rectangle extension. If the string "GL_ARB_texture_rectangle" is present in the EXTENSIONS string, as queried with GetString(), then the compiler will behave as if

```
    #extension GL_ARB_texture_rectangle : require
```

is present in the shader.

Add the following (previously reserved) keywords to the first part of section 3.6 on page 14:

```
sampler2DRect
sampler2DRectShadow
```

Add to section 8.7 "Texture Lookup Functions"

Syntax:

```
vec4 texture2DRect(sampler2DRect sampler, vec2 coord)
vec4 texture2DRectProj(sampler2DRect sampler, vec3 coord)
vec4 texture2DRectProj(sampler2DRect sampler, vec4 coord)
```

Description:

"Use the texture coordinate coord to do a texture lookup in the rectangle texture currently bound to sampler. For the projective ("Proj") version, the texture coordinate (coord.s, coord.t) is divided by the last component of coord. The third component of coord is ignored for the vec4 coord variant.

No "bias" parameter or "Lod" suffixed functions for rectangle textures are supported because mipmaps are not allowed for rectangular textures."

Syntax:

```
vec4 shadow2DRect(sampler2DRectShadow sampler, vec3 coord)
vec4 shadow2DRectProj(sampler2DRectShadow sampler, vec4 coord)
```

Description

"Use texture coordinate coord to do a depth comparison lookup on the rectangular depth texture bound to sampler, as described in section 3.8.14 of version 2.0 of the OpenGL specification. The 3rd component of coord (coord.p) is used as the R value. The texture bound to sampler must be a depth texture, or results are undefined. For the projective version ("Proj"), the texture coordinate (coord.s, coord.t, coord.p) is divided by the last component of coord, giving a R value of coord.p / coord.q.

No "bias" parameter or "Lod" suffixed functions for rectangle textures are supported because mipmaps are not allowed for rectangle textures."

Additions to the GLX Specification

None

GLX Protocol

None

Dependencies on OpenGL 1.4 and ARB_texture_mirrored_repeat

```
    If OpenGL 1.4 (or ARB_mirrored_repeat) is not supported, references
    to the MIRRORED_REPEAT (or MIRRORED_REPEAT_ARB) wrap mode in this
    document should be ignored.
```

Dependencies on ATI_texture_mirror_once

```
    If ATI_texture_mirror_once is not supported, references to the
    MIRROR_CLAMP_ATI and MIRROR_CLAMP_TO_EDGE_ATI wrap modes in this
    document should be ignored.
```

Dependencies on EXT_paletted_texture

```
    If EXT_paletted_texture is not supported, references to the
    COLOR_INDEX, COLOR_INDEX<n>_EXT, ColorTable, and ColorTableEXT
    should be ignored.
```

Dependencies on EXT_texture_compression_s3tc

```
    If EXT_texture_compression_s3tc is not supported, references
    to CompressedTexImage2D and CompressedTexSubImageARB and the
    COMPRESSED_*_S3TC_DXT*_EXT enumerants should be ignored.
```

Dependencies on EXT_texture_mirror_clamp

```
    If EXT_texture_mirror_clamp is not supported, references to the
    MIRROR_CLAMP_EXT, MIRROR_CLAMP_TO_EDGE_EXT, and
    MIRROR_CLAMP_TO_BORDER_EXT wrap modes in this document should be
    ignored.
```

Errors

```
    INVALID_ENUM is generated when ColorTable (or ColorTableEXT or the
    various ColorTable and ColorTableEXT alternative commands) is called
    and the target is TEXTURE_RECTANGLE_ARB or
    PROXY_TEXTURE_RECTANGLE_ARB.

    INVALID_ENUM is generated when TexImage2D is called and the target
    is TEXTURE_RECTANGLE_ARB or PROXY_TEXTURE_RECTANGLE_ARB and the
    format is COLOR_INDEX or the internalformat is COLOR_INDEX or one of
    the COLOR_INDEX<n>_EXT internal formats.

    INVALID_VALUE is generated when TexImage2D is called when the target
    is TEXTURE_RECTANGLE_ARB if border is any value other than zero or
    the level is any value other than zero.

    INVALID_VALUE is generated when TexImage2D is called when the target
    is TEXTURE_RECTANGLE_ARB if the width is less than zero or the
    height is less than zero.

    INVALID_VALUE is generated when TexSubImage2D or CopyTexSubImage2D
    is called when the target is TEXTURE_RECTANGLE_ARB if the level is
    any value other than zero.

    INVALID_ENUM is generated when one of the CompressedTexImage<n>D
    commands is called when the target parameter is
```

    TEXTURE_RECTANGLE_ARB or PROXY_TEXTURE_RECTANGLE_ARB.

    INVALID_ENUM is generated when one of the CompressedTexSubImage<n>D
    commands is called when the target parameter is TEXTURE_RECTANGLE_ARB
    or PROXY_TEXTURE_RECTANGLE_ARB.

    INVALID_ENUM is generated when TexParameter is called with a target
    of TEXTURE_RECTANGLE_ARB and the TEXTURE_WRAP_S, TEXTURE_WRAP_T,
    or TEXTURE_WRAP_R parameter is set to REPEAT, MIRRORED_REPEAT,
    MIRROR_CLAMP_ATI, or MIRROR_CLAMP_TO_EDGE_ATI.

    INVALID_ENUM is generated when TexParameter is called with a target
    of TEXTURE_RECTANGLE_ARB and the TEXTURE_MIN_FILTER is set to a
    value other than NEAREST or LINEAR.

    INVALID_VALUE is generated when TexParameter is called with a target
    of TEXTURE_RECTANGLE_ARB and the TEXTURE_BASE_LEVEL is set to any
    value other than zero.

    INVALID_VALUE is generated when GetTexImage is called with a lod not
    zero when the tex is TEXTURE_RECTANGLE_ARB.

New State

  - (Table 6.15, Texture Objects, pg. 241) amend/add the following entries:

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
TEXTURE_RECTANGLE_ARB	2* x B	IsEnabled	False	True if rectangular texturing is enabled	3.8.15	texture/enable
TEXTURE_BINDING_RECTANGLE_ARB	2* x Z+	GetIntegerv	0	Texture object for texture rectangle	3.8.11	texture
TEXTURE_RECTANGLE_ARB	n x I	GetTexImage	see 3.8	rectangular texture image for lod 0	3.8	-

  - (Table 6.16, Texture Objects (cont.), pg. 242) amend/add the following
entries:

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
TEXTURE_MIN_FILTER	n x Z6	GetTexParameter	See 3.8 except for rectangular which is LINEAR	Texture minification function	3.8.8	texture
TEXTURE_WRAP_S	n x Z5	GetTexParameter	REPEAT except for rectangular which is CLAMP_TO_EDGE	Texture wrap mode S	3.8.7	texture
TEXTURE_WRAP_T	n x Z5	GetTexParameter	REPEAT except for rectangular which is CLAMP_TO_EDGE	Texture wrap mode T (2D, 3D, cubemap, rectangle textures only)	3.8.7	texture
TEXTURE_WRAP_R	n x Z5	GetTexParameter	REPEAT except for rectangular which is CLAMP_TO_EDGE	Texture wrap mode R (3D textures only)	3.8.7	texture

New Implementation Dependent State

  - (Table 6.28, Implementation Dependent Values, pg. 254) add the following entry:

Get Value	Type	Get Command	Minimum Value	Description	Sec	Attribute
MAX_RECTANGLE_TEXTURE_SIZE_ARB	Z+	GetIntegerv	64	Maximum rectangular texture image dimension	3.8.1	-

Backwards Compatibility

This extension is semantically equivalent to EXT_texture_rectangle and NV_texture_rectangle. The tokens, and name strings now refer to ARB instead of EXT or NV. Enumerant values are unchanged.

Revision History

3/5/2004 - Updated page numbers and other numbers to reflect OpenGL 1.5; removed bogus "Convolution" language saying how glGetTexImage applies convolution (language was in 1.2.1 but removed in 1.3). ARB_texture_non_power_of_two and EXT_texture_mirror_clamp interactions added.

2/23/2005 - Fix the GLSL interaction: 1) GLSL functions require a vector (not scalar) parameter for the texture coordinate set: 2) The actual reserved types are sampler2DRect and sampler2DRectShadow (not samplerRect and samplerRectShadow); and 3) the shadow functions were missing.

7/8/2005 - Further fixes to GLSL interaction based on ARB meeting discussion: 1) Add OpenGL 2.0 language interaction for when shadow accesses are defined for rectangle textures; 2) add an issue to document the discussion; 3) bumped revision to 1.1; 4) documented GLSL preprocessor define; 5) documented sampler enums; and generally update the specification page numbers to be written against OpenGL 2.0. Also added to the contributors list.

7/15/2005 - This is revision 1.2.
1) Allow loading of DEPTH_COMPENENT textures for rectangular texture targets. 2) Switched some of the paramters ws, hs, ds for wt, ht, dt, and vice-versa to be in line with the cleanup already done in the OpenGL 2.0 specification. 3) Added issue 18. 4) Deleted the 'dependencies on ARB_texture_non_power_of_two' section since that is core OpenGL 2.0 functionality. 5) Removed some redundant language. 6) Added language describing changes to the GLSL spec explaining the #extension behavior. 7) Added to the contributors list and sorted it by last name.

10/4/2005 - Revision 1.21 - Whitespace cleanup

Name

    ARB_transpose_matrix

Name Strings

    GL_ARB_transpose_matrix

Status

    Complete. Approved by ARB on 12/8/1999

Version

    Last Modified Date: January 3, 2000
    Author Revision: 1.3

Number

    ARB Extension #3

Dependencies

    This extensions is written against the OpenGL 1.2 Specification.
    May be implemented in any version of OpenGL.

Overview

    New functions and tokens are added allowing application matrices
    stored in row major order rather than column major order to be
    transferred to the OpenGL implementation. This allows an application
    to use standard C-language 2-dimensional arrays (m[row][col]) and
    have the array indices match the expected matrix row and column indexes.
    These arrays are referred to as transpose matrices since they are
    the transpose of the standard matrices passed to OpenGL.

    This extension adds an interface for transfering data to and from the
    OpenGL pipeline, it does not change any OpenGL processing or imply any
    changes in state representation.

IP Status

    No IP is believed to be involved.

Issues

    * Why do this?

        It's very useful for layered libraries that desire to use two
        dimensional C arrays as matrices. It avoids having the layered
        library perform the transpose itself before calling OpenGL since
        most OpenGL implementations can efficiently perform the transpose
        while reading the matrix from client memory.

    * Why not add a mode?

        It's substantially more confusing and complicated to add a mode.

```
        Simply adding two new entry points saves considerable confusion
        and avoids having layered libraries need to query the current mode
        in order to send a matrix with the correct memory layout.

    * Why not a utility routine in GLU

        It costs some performance.  It is believed that most OpenGL
        implementations can perform the transpose in place with negligble
        performance penalty.

    * Why use the name transpose?

        It's sure a lot less confusing than trying to ascribe unambiguous
        meaning to terms like row and column.  It could be matrix_transpose
        rather than transpose_matrix though.

    * Short Transpose to Trans?
```

New Procedures and Functions

```
    void LoadTransposeMatrix{fd}ARB(T m[16]);
    void MultTransposeMatrix{fd}ARB(T m[16]);
```

New Tokens

```
    Accepted by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv,
    and GetDoublev

        TRANSPOSE_MODELVIEW_MATRIX_ARB    0x84E3
        TRANSPOSE_PROJECTION_MATRIX_ARB   0x84E4
        TRANSPOSE_TEXTURE_MATRIX_ARB      0x84E5
        TRANSPOSE_COLOR_MATRIX_ARB        0x84E6
```

Additions to Chapter 2 of the 1.2 OpenGL Specification (OpenGL Operation)

```
    Add to Section 2.10.2 Matrices  <before LoadIdentity>

    LoadTransposeMatrixARB takes a 4x4 matrix stored in row-major order as
    Let transpose(m,n) be defined as

        n[0] = m[0];
        n[1] = m[4];
        n[2] = m[8];
        n[3] = m[12];
        n[4] = m[1];
        n[5] = m[5];
        n[6] = m[9];
        n[7] = m[13];
        n[8] = m[2];
        n[9] = m[6];
        n[10] = m[10];
        n[11] = m[14];
        n[12] = m[3];
        n[13] = m[7];
        n[14] = m[11];
        n[15] = m[15];

    The effect of LoadTransposeMatrixARB(m) is then the same as the effect of
    the command sequence

        float n[16];
        transpose(m,n)
        LoadMatrix(n);

    The effect of MultTransposeMatrixARB(m) is then the same as the effect of
    the command sequence

        float n[16];
        transpose(m,n);
        MultMatrix(n);
```

Additions to Chapter 3 of the 1.2 OpenGL Specification (Rasterization)

```
    None
```

Additions to Chapter 4 of the 1.2 OpenGL Specification (Per-Fragment Operations and the Framebuffer)

```
    None
```

Additions to Chapter 5 of the 1.2 OpenGL Specification (Special Functions)

```
    None
```

Additions to Chapter 6 of the 1.2 OpenGL Specification (State and State Requests)

```
    Matrices are queried and returned in their transposed form by calling
    GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev with <pname> set to
    TRANSPOSE_MODELVIEW_MATRIX_ARB, TRANSPOSE_PROJECTION_MATRIX_ARB,
    TRANSPOSE_TEXTURE_MATRIX_ARB, or TRANSPOSE_COLOR_MATRIX_ARB.
    The effect of GetFloatv(TRANSPOSE_MODELVIEW_MATRIX_ARB,m) is then the same
    as the effect of the command sequence
```

```
        float n[16];
        GetFloatv(MODELVIEW_MATRIX_ARB,n);
        transpose(n,m);
```

    Similar results occur for TRANSPOSE_PROJECTION_MATRIX_ARB,
    TRANSPOSE_TEXTURE_MATRIX_ARB, and TRANSPOSE_COLOR_MATRIX_ARB.

Additions to Appendix A of the OpenGL 1.2.1 Specification (Invariance)

None

Additions to the GLX Specification

None

GLX Protocol

LoadTransposeMatrix and MultTransposeMatrix are layered on top of LoadMatrix and MultMatrix protocol performing client-side translation. The Get commands are passed over the wire as part of the generic Get protocol with no translation required.

Errors

No new errors, but error behavoir is inherited by the commands that the transpose commands are implemented on top of (LoadMatrix, MultMatrix, and Get*).

New State

None

TRANSPOSE_*_MATRIX_ARB refer to the same state as their non-transposed counterparts.

New Implementation Dependent State

None

Revision History

```
    * Revision 1.1 - initial draft (18 Mar 1999)
    * Revision 1.2 - changed to use layered specification and ARB affix
                     (23 Nov 1999)
    * Revision 1.3 - Minor tweaks to GLX protocol and Errors. (7 Dec 1999)
```

Conformance Testing

Load and Multiply the modelview matrix (initialized to identity each time) using LoadTransposeMatrixfARB and MultTransposeMatrixfARB with the matrix:

```
        ( 1  2  3  4 )
        ( 5  6  7  8 )
        ( 9 10 11 12 )
        (13 14 15 16 )

    and get the modelview matrix using TRANSPOSE_MODELVIEW_MATRIX_ARB and
    validate that the matrix is correct.  Get the matrix using
    MODELVIEW_MATRIX and verify that it is the transpose of the above
    matrix.  Load and Multiply the modelview matrix using LoadMatrixf
    and MultMatrixf with the above matrix and verify that the correct
    matrix is on the modelview stack using gets of MODELVIEW_MATRIX
    and TRANSPOSE_MODELVIEW_MATRIX_ARB.
```

Name

    ARB_vertex_buffer_object

Name Strings

    GL_ARB_vertex_buffer_object

IP Status

    None.

Status

    Complete. Approved by ARB on February 12, 2003.

Version

    Last Modified Date: January 21, 2003
    Revision: 0.91

Number

    ARB Extension #28

Dependencies

    Written based on the wording of the OpenGL 1.4 specification.

    GL_ARB_vertex_blend affects the definition of this extension.

    GL_ARB_vertex_program affects the definition of this extension.

    GL_EXT_vertex_shader affects the definition of this extension.

Overview

    This extension defines an interface that allows various types of data
    (especially vertex array data) to be cached in high-performance
    graphics memory on the server, thereby increasing the rate of data
    transfers.

    Chunks of data are encapsulated within "buffer objects", which
    conceptually are nothing more than arrays of bytes, just like any
    chunk of memory. An API is provided whereby applications can read
    from or write to buffers, either via the GL itself (glBufferData,
    glBufferSubData, glGetBufferSubData) or via a pointer to the memory.

    The latter technique is known as "mapping" a buffer. When an
    application maps a buffer, it is given a pointer to the memory. When
    the application finishes reading from or writing to the memory, it is
    required to "unmap" the buffer before it is once again permitted to
    use that buffer as a GL data source or sink. Mapping often allows
    applications to eliminate an extra data copy otherwise required to
    access the buffer, thereby enhancing performance. In addition,
    requiring that applications unmap the buffer to use it as a data
    source or sink ensures that certain classes of latent synchronization

bugs cannot occur.

Although this extension only defines hooks for buffer objects to be used with OpenGL's vertex array APIs, the API defined in this extension permits buffer objects to be used as either data sources or sinks for any GL command that takes a pointer as an argument. Normally, in the absence of this extension, a pointer passed into the GL is simply a pointer to the user's data. This extension defines a mechanism whereby this pointer is used not as a pointer to the data itself, but as an offset into a currently bound buffer object. The buffer object ID zero is reserved, and when buffer object zero is bound to a given target, the commands affected by that buffer binding behave normally. When a nonzero buffer ID is bound, then the pointer represents an offset.

In the case of vertex arrays, this extension defines not merely one binding for all attributes, but a separate binding for each individual attribute. As a result, applications can source their attributes from multiple buffers. An application might, for example, have a model with constant texture coordinates and variable geometry. The texture coordinates might be retrieved from a buffer object with the usage mode "STATIC_DRAW", indicating to the GL that the application does not expect to update the contents of the buffer frequently or even at all, while the vertices might be retrieved from a buffer object with the usage mode "STREAM_DRAW", indicating that the vertices will be updated on a regular basis.

In addition, a binding is defined by which applications can source index data (as used by DrawElements, DrawRangeElements, and MultiDrawElements) from a buffer object. On some platforms, this enables very large models to be rendered with no more than a few small commands to the graphics device.

It is expected that a future extension will allow sourcing pixel data from and writing pixel data to a buffer object.

Issues

What should this extension be called?

RESOLVED: By unanimous consent among the working group members, the name was chosen to be "ARB_vertex_buffer_object". A large number of other names were considered throughout the lifetime of the proposal, especially "vertex_array_object" (originally), "buffer_object" (later on), and "memory_object" (near the end), but the name "vertex_buffer_object" was ultimately chosen.

In particular, this name emphasizes not only that we have created a new type of object that encapsulates arbitrary data (buffer objects), but also, in particular, that these objects are used in this extension to source vertex data. The name also is intentionally similar to "vertex buffers", although it should be emphasized that there is no such thing as a "vertex buffer" in the terminology of this extension. The term "buffer object" is the correct noun.

How is this extension different from ATI_vertex_array_object plus ATI_map_object_buffer?

The following summarizes the major differences.
- VAOs renamed to "buffer objects", to signify that they can be used for more than just vertex data. Other renaming and API changes to try to better match OpenGL conventions.
- The standard GL pointer APIs have been overloaded to be able to refer to offsets within these buffers, rather than adding new entry points.
- The usage modes permitted for buffers have been augmented significantly, to reflect a broader class of application behaviors.
- A new entry point allows reading back the contents of a buffer object.

How is this extension different from NV_vertex_array_range?

The following summarizes the major differences.
- Applications are no longer responsible for memory management and synchronization.
- Applications may still access high-performance memory, but this is optional, and such access is more restricted.
- Buffer changes (glBindBufferARB) are generally expected to be very lightweight, rather than extremely heavyweight (glVertexArrayRangeNV).
- A platform-specific allocator such as wgl/glXAllocateMemoryNV is no longer required.

How does this extension relate to NV_pixel_data_range?

A future extension could be created based on the framework created here that would support analogous functionality to that provided by NV_pixel_data_range. Presumably, this extension would require little more than two new targets for BindBuffer, named (say) UNPACK_PIXELS and PACK_PIXELS. The lists of commands affected by these bindings could easily be taken verbatim out of the NV_pixel_data_range specification.

Should this extension include support for allowing vertex indices to be stored in buffer objects?

RESOLVED: YES. It is easily and cleanly added with just the addition of a binding point for the index buffer object. Since our approach of overloading pointers works for any pointer in GL, no additional APIs need be defined, unlike in the various *_element_array extensions.

Note that it is expected that implementations may have different memory type requirements for efficient storage of indices and vertices. For example, some systems may prefer indices in AGP memory and vertices in video memory, or vice versa; or, on systems where DMA of index data is not supported, index data must be stored in (cacheable) system memory for acceptable performance. As a result, applications are strongly urged to put their models' vertex and index data in separate buffers, to assist drivers in choosing the most efficient locations.

Should the layout of an array store be defined at array store creation time?

RESOLVED: NO. This could provide better performance if the client specifies a data type that the hardware doesn't support, but this isn't a performance path anyways, and it adds a cumbersome interface on top of the extension.

Should there be some sort of scheme for allowing applications to stream vertex data efficiently?

RESOLVED: YES. Applications that generate their data on the fly end up doing an extra data copy unless they are given a pointer into memory that the graphics hardware can DMA from. The performance win from doing this can be significant.

Should the client be able to retrieve a pointer to a buffer object?

RESOLVED: YES. This solves the previous problem. Since GL vertex array formats are already user-visible, this does not suffer from the sorts of formatting issues that would arise if the GL allowed applications to retrieve pointers to texture objects or to the framebuffer. Synchronization can be a concern, but proper usage of this extension will minimize its overhead.

Should this extension sit on top of the existing vertex array implementation, instead of introducing a new set of API calls?

RESOLVED: YES. This simplifies the API, and separating out the buffer binding from the offset/stride within the buffer leads to an elegant "BindBufferARB" command that can be used for other parts of GL like the pixel path.

Should buffer object state overlap with existing vertex array pointer state, or should there be new drawing commands, e.g., DrawArrayObject?

RESOLVED: OVERLAP. The exponential growth in drawing commands is problematic. Even without this, there is already ArrayElement, DrawArrays, DrawElements, DrawRangeElements, MultiDrawArrays, and MultiDrawElements.

Does the buffer binding state push/pop?

RESOLVED: YES. It pushes/pops on the client with the rest of the vertex array state. Some revisions of the ATI VAO spec listed a push/pop attrib "objbuf", but no new bit was defined; all this has been moved into the standard "vertex-array" bit.

Note that both the user-controlled binding ARRAY_BUFFER_ARB binding point and the per-array bindings push and pop.

Note that additional binding points, such as ones for pixel or texture transfers, would not be part of the vertex array state, and thus would likely push and pop as part of the pixel store (client) state when they are defined.

How is the decision whether to use the array pointer as an offset or as a real pointer made?

RESOLVED: When the default buffer object (object zero) is bound, all pointers behave as real pointers. When any other object is bound, all pointers are treated as offsets. Conceptually, one can imagine that buffer object zero is a buffer object sitting at base NULL and with an extent large enough that it covers all of the system's virtual address space.

Note that this approach essentially requires that binding points be client (not server) state.

Can buffer objects be shared between contexts in the same way that display lists are?

RESOLVED: YES. All potentially large OpenGL objects, such as display lists and textures, can be shared, and this is an important capability. Note, however, that sharing requires that buffer objects be server (not client) state, since it is not possible to share client state.

Should buffer objects be client state or server state?

RESOLVED: Server state. Arguments for client state include:

- Buffer data are stored in client-side format, making server storage complex when client and server endianness differ.
- Vertex arrays are client state.

These arguments are outweighed by the significant advantages of server state, including:

- Server state can be shared between contexts, and this is expected to be an important capability (sharing of texture objects is very common).
- In the case of indirect rendering, performance may be very significantly greater for data stored on the server side of the wire.

How is synchronization enforced when buffer objects are shared by multiple OpenGL contexts?

RESOLVED: It is generally the clients' responsibility to synchronize modifications made to shared buffer objects. GL implementations will make some effort to avoid deletion of in-use buffer objects, but may not be able to ensure this handling.

What happens if a currently bound buffer object is deleted?

RESOLVED: Orphan. To avoid chasing invalid pointers OpenGL implementations will attempt to defer the deletion of any buffer object until that object is not bound by any client in the share list. It should be possible to implement this behavior efficiently in the direct rendering case, but the implementation may be difficult/impossible in the indirect rendering case.

Since synchronization during sharing is a client responsibility, this behavior is acceptable.

Should there be a way to query the data in a buffer object?

RESOLVED: YES. Almost all objects in OpenGL are fully queriable, and since these objects are simply byte arrays, there does not seem to be any reason to do things otherwise here. The primary exceptions to GL queriability are cases where such functionality would be extremely burdensome to provide, as is the case with display lists.

Do buffer objects survive screen resolution changes, etc.?

RESOLVED: YES. This is not mentioned in the spec, so by default they behave just like other OpenGL state, like texture objects -- the data is unmodified by external events like modeswitches, switching the system into standby or hibernate mode, etc.

What happens to a mapped buffer when a screen resolution change or other such window-system-specific system event occurs?

RESOLVED: The buffer's contents may become undefined. The application will then be notified at Unmap time that the buffer's contents have been destroyed. However, for the remaining duration of the map, the pointer returned from Map must continue to point to valid memory, in order to ensure that the application cannot crash if it continues to read or write after the system event has been handled.

What happens to the pointer returned by MapBufferARB after a call to UnmapBufferARB?

RESOLVED: The pointer becomes totally invalid. Note that drivers are free to move the underlying buffer or even unmap the memory, leaving the virtual addresses in question pointing at nothing. Such flexibility is necessary to enable efficient implementations on systems with no virtual memory; with limited control over virtual memory from graphics drivers; or where virtual address space is at a premium.

How does indirect rendering work?

It is not currently specified, but the basic planned outline is as follows.

All of the object management commands -- Gen, Is, Delete -- go to the server immediately with normal protocol. So does Bind. However, when someone does an implicit bind via one of the pointer commands (e.g. VertexPointer), the server may not necessarily be notified immediately of the new object bound to the (in this case) VERTEX_ARRAY_BUFFER_BINDING.

BufferData and BufferSubData are sent over the wire just as TexImage2D and TexSubImage2D, and GetBufferSubData does a round trip, just like GetTexImage. MapBuffer goes over the wire with a request to map; the server replies to tell the client whether the map succeeded or failed, and the client returns a pointer to a system memory buffer in the event of success. If the map is readable, the server passes back the contents of the buffer, while if the map is writeable, at Unmap time, the client passes back the new contents. Unmap would always return TRUE.

Both GetBufferParameteriv and GetBufferPointerv go to the server.

Whenever the application sources data from a buffer object, several new protocols are defined to specify where to obtain the data from. One new command might be called "BindArray", which would have arguments <array>, <buffer>, offset>, <type>, <size>, <stride>, and <normalized>. <array> might be VERTEX_ARRAY, NORMAL_ARRAY, etc. <buffer> would be the ID of the buffer object to be used as source, or zero if no buffer object. <offset> would be a 64-bit (?) integer. <type>, <size>, <stride>, and <normalized> would all be the same as the various arguments to the *Pointer commands. Another new command might be "ArrayElementServer", which would dereference all arrays with a nonzero <buffer> on the server side, just as if immediate mode had been used. If only some arrays were coming from buffer objects and some from user memory, the client would dereference the ones in user memory and pass them in as immediate mode protocol.

If all arrays came from the server, additional optimized APIs could be provided. A "DrawArraysServer" and "DrawElementsServer" would be cheaper than a sequence of "ArrayElementServer" commands. For indices coming from a buffer object, a "DrawElementArrayServer" might be added.

At initialization time, the client and server would exchange a handshake to see if the server can understand the client's storage of the various GL data types. It is expected that nearly all clients and servers would use just two data type representations, namely, "standard little endian with IEEE floats" and "standard big endian with IEEE floats".

Are any of these commands allowed inside Begin/End?

RESOLVED: NO, with the possible exception of BindBuffer, which should not be used inside a Begin/End but will have undefined error behavior, like most vertex array commands.

What happens when an attempt is made to access data outside the bounds of the buffer object with a command that dereferences the arrays?

RESOLVED: ALLOW PROGRAM TERMINATION. In the event of a software fallback, bounds checking can become impractical. Since applications don't know the actual address of the buffer object and only provide an offset, they can't ever guarantee that out-of-bounds offsets will fall on valid memory. So it's hard to do any better than this.

Of course, such an event should not be able to bring down the system, only terminate the program.

What type should <offset> and <size> arguments use?

RESOLVED: We define new types that will work well on 64-bit systems, analogous to C's "intptr_t". The new type "GLintptrARB" should be used in place of GLint whenever it is expected that values might exceed 2 billion. The new type "GLsizeiptrARB" should be used in place of GLsizei whenever it is expected that counts might exceed 2 billion. Both types are defined as signed integers large enough to contain any pointer value. As a result, they naturally scale to larger numbers of bits on systems with 64-bit or even larger pointers.

The offsets introduced in this extension are typed GLintptrARB, consistent with other GL parameters that must be non-negative, but are arithmetic in nature (not uint), and are not sizes; for example, the xoffset argument to TexSubImage*D is of type GLint. Buffer sizes are typed GLsizeiptrARB.

The idea of making these types unsigned was considered, but was ultimately rejected on the grounds that supporting buffers larger than 2 GB was not deemed important on 32-bit systems.

Should buffer maps be client or server state?

RESOLVED: Server. If a buffer is being shared by multiple clients, it will also be desirable to share the mappings of that buffer. In cases where the mapping cannot shared (for example, in the case of indirect rendering) queries of the map pointer by clients other than the one that created the map will return a null pointer.

Should "Unmap" be treated as one word or two?

RESOLVED: One word.

Should "usage" be a parameter to BufferDataARB, or specified separately using a parameter specification command, e.g., BufferParameteriARB?

RESOLVED: Parameter to BufferDataARB. It is desirable for the implementation to know the usage when the buffer is initialized, so including it in the initialization command makes sense. This avoids manpage notes such as "Please specify the usage before you initialize the buffer".

Should it be possible to change the usage of an initialized buffer?

RESOLVED: NO. Unless it is shown that this flexibility is necessary, it will be easier for implementations to be efficient if usage cannot be changed. (Except by re-initializing the buffer completely.)

Should we allow for the possibility of multiple simultaneous maps for a single buffer?

RESOLVED: NO. If multiple maps are allowed, the mapping semantics become very difficult to understand and to specify. It is also unclear that there are any benefits to offering such functionality. Therefore, only one map per buffer is permitted.

Note: the limit of one map per buffer eliminates any need for "sub-region" mapping. The single map always maps the entire data store of the buffer.

Should it be an error to render from a currently mapped buffer?

RESOLVED: YES. Making this an error rather than undefined makes the API much more bulletproof.

Should it be possible for the application to query the "viability" of the data store of a buffer?

RESOLVED: NO. UnmapBuffer can return FALSE to indicate this, but there is no additional query to check whether the data has been lost. In general, most/all GL state is queriable, unless there is a compelling reason otherwise. However, on examination, it appears that there are several compelling reasons otherwise in this case. In particular, the default for this state variable is non-obvious (is the data "valid" when no data has been specified yet?), and it's unclear when it should be reset (BufferData only? BufferSubData? A successful UnmapBuffer?). After these issues came to light, the query was removed from the spec.

What should the error behavior of BufferDataARB and MapBufferARB be?

RESOLVED: BufferDataARB returns no value and sets OUT_OF_MEMORY if the buffer could not be created, whereas MapBufferARB returns NULL and also sets OUT_OF_MEMORY if the buffer could not be mapped.

Should UnmapBufferARB return a boolean indicating data integrity?

RESOLVED: YES, since the Unmap is precisely the point at which the buffer can no longer be lost.

How is unaligned data handled?

RESOLVED: All client restrictions on data alignment must be met, and in addition to that, all offsets must be multiples of the size of the underlying data type. So, for example, float data in a buffer object must have an offset that is (typically) a multiple of 4. This should make the server implementation easier, since this additional rule will guarantee that no alignment checking is required on most platforms.

Should MapBufferARB return the pointer to the map, or should there be a separate call to ask for the pointer?

RESOLVED: BOTH. For convenience, MapBufferARB returns a pointer or NULL in the event of failure; but since most/all GL state is queriable, you can also query the pointer at a later point in time. If the buffer is not mapped, querying the pointer should return NULL.

Should there be one binding point for all arrays or several binding points, one for each array?

RESOLVED: One binding point for all arrays. Index data uses a separate target.

Should there be a PRESERVE/DISCARD option on BufferSubDataARB? On MapBufferARB?

RESOLVED: NO, NO. ATI_vertex_array_object had this option for UpdateObjectBufferATI, which is the equivalent of BufferSubDataARB, but it's unclear whether this has any utility. There might be some utility for MapBufferARB, but forcing the user to call BufferDataARB again with a NULL data pointer has some advantages of its own, such as forcing the user to respecify the size.

Should there be an option for MapBufferARB that guarantees nonvolatile memory?

RESOLVED: NO. On systems where volatile memory spaces are a concern, there is little or no way to supply nonvolatile memory without crippling performance badly. In some cases, it might not even be possible to implement Map except by returning system memory. Systems that do not have problems with volatility are, of course, welcome to return TRUE from UnmapBufferARB all the time. If applications want the ease of use that results from not having to check for lost data, they can still use BufferDataARB and BufferSubDataARB, so the burden is not too great.

What new usages do we need to add?

RESOLVED. We have defined a 3x3 matrix of usages. The pixel-related terms draw, read, and copy are used to distinguish between three basic data paths: application to GL (draw), GL to application (read), and GL to GL (copy). The terms stream, static, and dynamic are used to identify three data access patterns: specify once and use once or perhaps only a few times (stream), specify once and use many times (static), and specify and use repeatedly (dynamic).

Note that the "copy" and "read" usage token values will become meaningful only when pixel transfer capability is added to buffer objects by a (presumed) subsequent extension.

Note that the data paths "draw", "read", and "copy" are analogous in both name and meaning to the GL commands DrawPixels, ReadPixels, and CopyPixels, respectively.

Is it legal C to use pointers as offsets?

We haven't come to any definitive conclusion about this. The proposal is to convert to pointer as:

```
pointer = (char *)NULL + offset;
```

And convert back to offset as:

```
offset = (char *)pointer - (char *)NULL;
```

Varying opinions have been expressed as to whether this is legal, although no one could provide an example of a real system where any problems would occur.

Should we add new Offset commands, e.g., VertexOffset, if the pointer approach has some compatibility concerns?

RESOLVED: NO. The working group voted that the existing pointer-as-offset approach is acceptable.

Which commands are compiled into display lists?

RESOLVED: None of the commands in this extension are compiled into display lists. The reasoning is that the server may not have up-to-date buffer bindings, since BindBuffer is a client command.

Just as without this extension, vertex data is dereferenced when ArrayElement, etc. are compiled into a display list.

Should there be a new command "DiscardAndMapBuffer" that is equivalent to BufferDataARB with NULL pointer followed by MapBufferARB?

RESOLVED: NO, no one has come up with a clearly superior proposal that everyone can agree on.

Are any GL commands disallowed when at least one buffer object is mapped?

RESOLVED: NO. In general, applications may use whatever GL commands they wish when a buffer is mapped. However, several other restrictions on the application do apply: the application must not attempt to source data out of, or sink data into, a currently mapped buffer. Furthermore, the application may not use the pointer returned by Map as an argument to a GL command. (Note that this last restriction is unlikely to be enforced in practice, but it violates reasonable expectations about how the extension should be used, and it doesn't seem to be a very interesting usage model anyhow. Maps are for the user, not for the GL.)

More restrictive rules were considered (for example, "after calling MapBuffer, all GL commands except for UnmapBuffer produce errors"), but this was considered far too restrictive. The expectation is that an application might map a buffer and start filling it in a different thread, but continue to render in its main thread (using a different buffer or no buffer at all). So no commands are disallowed simply because a buffer happens to be mapped.

Should the usage and data arguments to BufferDataARB be swapped?

RESOLVED: NO. This would be more consistent with other things in GL if they were swapped, but no one seems to care. If we had caught this earlier, maybe, but it's just too late.

How does MultiDrawElements work?

The language gets a little confusing, but I believe it is quite clearly specified in the end. The argument <indices> to MultiDrawElements, which is of type "const void **", is an honest-to-goodness pointer to regular old system memory, no matter whether a buffer is bound or not. That memory in turn consists of an array of <primcount> pointers. If no buffer is bound, each of those <primcount> pointers is a regular pointer. If a buffer is bound, each of those <primcount> pointers is a fake pointer that represents an offset in the buffer object.

If you wanted to put the array of <primcount> offsets in a buffer object, you'd have to define a new extension with a new target.

When is the binding between a buffer object and a specific vertex array (e.g., VERTEX_ARRAY_BUFFER_BINDING_ARB) established?

The array's buffer binding is set when the array pointer is specified. Using the vertex array as an example, this is when VertexPointer is called. At that time, the current array buffer binding is used for the vertex array. The current array buffer binding is set by calling BindBufferARB with a <target> of ARRAY_BUFFER_ARB. Changing the current array buffer binding does not affect the bindings used by already established arrays.

```
  BindBufferARB(ARRAY_BUFFER_ARB, 1);
  VertexPointer(...);   // vertex array data points to buffer 1
  BindBufferARB(ARRAY_BUFFER_ARB, 2);
  // vertex array data still points to buffer 1
```

New Procedures and Functions

```
    void BindBufferARB(enum target, uint buffer);
    void DeleteBuffersARB(sizei n, const uint *buffers);
    void GenBuffersARB(sizei n, uint *buffers);
    boolean IsBufferARB(uint buffer);

    void BufferDataARB(enum target, sizeiptrARB size, const void *data,
                       enum usage);
    void BufferSubDataARB(enum target, intptrARB offset, sizeiptrARB size,
                          const void *data);
    void GetBufferSubDataARB(enum target, intptrARB offset,
                             sizeiptrARB size, void *data);

    void *MapBufferARB(enum target, enum access);
    boolean UnmapBufferARB(enum target);

    void GetBufferParameterivARB(enum target, enum pname, int *params);
    void GetBufferPointervARB(enum target, enum pname, void **params);
```

New Tokens

Accepted by the <target> parameters of BindBufferARB, BufferDataARB, BufferSubDataARB, MapBufferARB, UnmapBufferARB, GetBufferSubDataARB, GetBufferParameterivARB, and GetBufferPointervARB:

```
        ARRAY_BUFFER_ARB                                0x8892
        ELEMENT_ARRAY_BUFFER_ARB                        0x8893
```

Accepted by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

```
        ARRAY_BUFFER_BINDING_ARB                        0x8894
        ELEMENT_ARRAY_BUFFER_BINDING_ARB                0x8895
        VERTEX_ARRAY_BUFFER_BINDING_ARB                 0x8896
        NORMAL_ARRAY_BUFFER_BINDING_ARB                 0x8897
        COLOR_ARRAY_BUFFER_BINDING_ARB                  0x8898
        INDEX_ARRAY_BUFFER_BINDING_ARB                  0x8899
        TEXTURE_COORD_ARRAY_BUFFER_BINDING_ARB          0x889A
        EDGE_FLAG_ARRAY_BUFFER_BINDING_ARB              0x889B
        SECONDARY_COLOR_ARRAY_BUFFER_BINDING_ARB        0x889C
        FOG_COORDINATE_ARRAY_BUFFER_BINDING_ARB         0x889D
        WEIGHT_ARRAY_BUFFER_BINDING_ARB                 0x889E
```

Accepted by the <pname> parameter of GetVertexAttribivARB:

```
        VERTEX_ATTRIB_ARRAY_BUFFER_BINDING_ARB          0x889F
```

Accepted by the <usage> parameter of BufferDataARB:

STREAM_DRAW_ARB	0x88E0
STREAM_READ_ARB	0x88E1
STREAM_COPY_ARB	0x88E2
STATIC_DRAW_ARB	0x88E4
STATIC_READ_ARB	0x88E5
STATIC_COPY_ARB	0x88E6
DYNAMIC_DRAW_ARB	0x88E8
DYNAMIC_READ_ARB	0x88E9
DYNAMIC_COPY_ARB	0x88EA

Accepted by the <access> parameter of MapBufferARB:

READ_ONLY_ARB	0x88B8
WRITE_ONLY_ARB	0x88B9
READ_WRITE_ARB	0x88BA

Accepted by the <pname> parameter of GetBufferParameterivARB:

BUFFER_SIZE_ARB	0x8764
BUFFER_USAGE_ARB	0x8765
BUFFER_ACCESS_ARB	0x88BB
BUFFER_MAPPED_ARB	0x88BC

Accepted by the <pname> parameter of GetBufferPointervARB:

BUFFER_MAP_POINTER_ARB	0x88BD

Additions to Chapter 2 of the 1.4 Specification (OpenGL Operation)

Add to Table 2.2:

"GL Type	Minimum Bit Width	Description
intptrARB	<ptrbits>	signed 2's complement binary integer
sizeiptrARB	<ptrbits>	Non-negative binary integer size"

Add to the paragraph under Table 2.2:

"<ptrbits> is the number of bits required to represent a pointer type; in other words, types intptrARB and sizeiptrARB must be sufficiently large as to store any address."

Add a new section "Buffer Objects" between sections 2.8 and 2.9:

"2.8A Buffer Objects

The vertex data arrays described in section 2.8 are stored in client memory. It is sometimes desirable to store frequently used client data, such as vertex array data, in high-performance server memory. GL buffer objects provide a mechanism that clients can use to allocate, initialize, and render from such memory.

The name space for buffer objects is the unsigned integers, with zero reserved for the GL. A buffer object is created by binding an unused name to ARRAY_BUFFER_ARB. The binding is effected by calling

```
    void BindBufferARB(enum target, uint buffer);
```

with <target> set to ARRAY_BUFFER_ARB and <buffer> set to the unused name. The resulting buffer object is a new state vector, initialized with a zero-sized memory buffer, and comprising the state values listed in Table BufObj1.

Name	Type	Initial Value	Legal Values
BUFFER_SIZE_ARB	integer	0	any non-negative integer
BUFFER_USAGE_ARB	enum	STATIC_DRAW_ARB	STREAM_DRAW_ARB, STREAM_READ_ARB, STREAM_COPY_ARB, STATIC_DRAW_ARB, STATIC_READ_ARB, STATIC_COPY_ARB, DYNAMIC_DRAW_ARB, DYNAMIC_READ_ARB, DYNAMIC_COPY_ARB
BUFFER_ACCESS_ARB	enum	READ_WRITE_ARB	READ_ONLY_ARB, WRITE_ONLY_ARB, READ_WRITE_ARB
BUFFER_MAPPED_ARB	boolean	FALSE	TRUE, FALSE
BUFFER_MAP_POINTER_ARB	void*	NULL	address

Table BufObj1: Buffer object parameters and their values.

BindBufferARB may also be used to bind an existing buffer object. If the bind is successful no change is made to the state of the newly bound buffer object, and any previous binding to <target> is broken.

While a buffer object is bound, GL operations on the target to which it is bound affect the bound buffer object, and queries of the target to which a buffer object is bound return state from the bound object.

In the initial state the GL-reserved name zero is bound to ARRAY_BUFFER_ARB. There is no buffer object corresponding to the name zero, so client attempts to modify or query buffer object state for the target ARRAY_BUFFER_ARB while zero is bound will generate GL errors.

Buffer objects are deleted by calling

```
    void DeleteBuffersARB(sizei n, const uint *buffers);
```

<buffers> contains <n> names of buffer objects to be deleted. After a buffer object is deleted it has no contents, and its name is again unused. Unused names in <buffers> are silently ignored, as is the value zero.

```
    The command

        void GenBuffersARB(sizei n, uint *buffers);

    returns <n> previously unused buffer object names in <buffers>.
    These names are marked as used, for the purposes of GenBuffersARB
    only, but they acquire buffer state only when they are first bound,
    just as if they were unused.

    While a buffer object is bound, any GL operations on that object
    affect any other bindings of that object.  If a buffer object is
    deleted while it is bound, all bindings to that object in the current
    context (i.e. in the thread that called DeleteBuffers) are reset to
    bindings to buffer zero.  Bindings to that buffer in other contexts
    and other threads are not affected, but attempting to use a deleted
    buffer in another thread produces undefined results, including but
    not limited to possible GL errors and rendering corruption.  Using a
    deleted buffer in another context or thread may not, however, result
    in program termination.

    The data store of a buffer object is created and initialized by
    calling

        void BufferDataARB(enum target, sizeiptrARB size,
                           const void *data, enum usage);

    with <target> set to ARRAY_BUFFER_ARB, <size> set to the size of the
    data store in basic machine units, and <data> pointing to the
    source data in client memory.  If <data> is non-null, then the source
    data is copied to the buffer object's data store.  If <data> is null,
    then the contents of the buffer object's data store are undefined.

    <usage> is specified as one of nine enumerated values, indicating
    the expected application usage pattern of the data store.  The
    values are:

        STREAM_DRAW_ARB    The data store contents will be specified once
                           by the application, and used at most a few
                           times as the source of a GL (drawing) command.
        STREAM_READ_ARB    The data store contents will be specified once
                           by reading data from the GL, and queried at
                           most a few times by the application.
        STREAM_COPY_ARB    The data store contents will be specified once
                           by reading data from the GL, and used at most
                           a few times as the source of a GL (drawing)
                           command.
        STATIC_DRAW_ARB    The data store contents will be specified once
                           by the application, and used many times as the
                           source for GL (drawing) commands.
        STATIC_READ_ARB    The data store contents will be specified once
                           by reading data from the GL, and queried many
                           times by the application.
        STATIC_COPY_ARB    The data store contents will be specified once
                           by reading data from the GL, and used many
                           times as the source for GL (drawing) commands.
```

DYNAMIC_DRAW_ARB	The data store contents will be respecified repeatedly by the application, and used many times as the source for GL (drawing) commands.
DYNAMIC_READ_ARB	The data store contents will be respecified repeatedly by reading data from the GL, and queried many times by the application.
DYNAMIC_COPY_ARB	The data store contents will be respecified repeatedly by reading data from the GL, and used many times as the source for GL (drawing) commands.

<usage> is provided as a performance hint only. The specified usage value does not constrain the actual usage pattern of the data store.

BufferDataARB deletes any existing data store, and sets the values of the buffer object's state variables to:

Name	Value
BUFFER_SIZE_ARB	<size>
BUFFER_USAGE_ARB	<usage>
BUFFER_ACCESS_ARB	READ_WRITE_ARB
BUFFER_MAPPED_ARB	FALSE
BUFFER_MAP_POINTER_ARB	NULL

Clients must align data elements consistent with the requirements of the client platform, with an additional base-level requirement that an offset within a buffer to a datum comprising N basic machine units be a multiple of N.

If the GL is unable to create a data store of the requested size, the error OUT_OF_MEMORY is generated.

To modify some or all of the data contained in a buffer object's data store, the client may use the command

```
    void BufferSubDataARB(enum target, intptrARB offset,
                          sizeiptrARB size, const void *data);
```

with <target> set to ARRAY_BUFFER_ARB. <offset> and <size> indicate the range of data in the buffer object that is to be replaced, in terms of basic machine units. <data> specifies a region of client memory <size> basic machine units in length, containing the data that replace the specified buffer range. An error is generated if <offset> or <size> is less than zero, or if <offset> + <size> is greater than the value of BUFFER_SIZE_ARB.

The entire data store of a buffer object can be mapped into the client's address space by calling

```
    void *MapBufferARB(enum target, enum access);
```

with <target> set to ARRAY_BUFFER_ARB. If the GL is able to map the buffer object's data store into the client's address space, MapBufferARB returns the pointer value to the data store. Otherwise MapBufferARB returns NULL, and the error OUT_OF_MEMORY is generated. <access> is specified as one of READ_ONLY_ARB, WRITE_ONLY_ARB, or

```
    READ_WRITE_ARB, indicating the operations that the client may perform
    on the data store through the pointer while the data store is mapped.

    MapBufferARB sets the following buffer object state values:

        Name                   Value
        ----                   -----
        BUFFER_ACCESS_ARB      <access>
        BUFFER_MAPPED_ARB      TRUE
        BUFFER_MAP_POINTER_ARB pointer to the data store

    It is an INVALID_OPERATION error to map a buffer data store that is
    in the mapped state.

    Non-null pointers returned by MapBufferARB may be used by the client
    to modify and query buffer object data, consistent with the access
    rules of the mapping, while the mapping remains valid.  No GL error
    is generated if the pointer is used to attempt to modify a
    READ_ONLY_ARB data store, or to attempt to read from a WRITE_ONLY_ARB
    data store, but operation may be slow and system errors (possibly
    including program termination) may result.  Pointer values returned
    by MapBufferARB may not be passed as parameter values to GL commands.
    For example, they may not be used to specify array pointers, or to
    specify or query pixel or texture image data; such actions produce
    undefined results, although implementations may not check for such
    behavior for performance reasons.

    It is an INVALID_OPERATION error to call BufferSubDataARB to modify
    the data store of a mapped buffer.

    Mappings to the data stores of buffer objects may have nonstandard
    performance characteristics.  For example, such mappings may be
    marked as uncacheable regions of memory, and in such cases reading
    from them may be very slow.  To ensure optimal performance, the
    client should use the mapping in a fashion consistent with the values
    of BUFFER_USAGE_ARB and BUFFER_ACCESS_ARB.  Using a mapping in a
    fashion inconsistent with these values is liable to be multiple
    orders of magnitude slower than using normal memory.

    After the client has specified the contents of a mapped data store,
    and before the data in that store are dereferenced by any GL commands,
    the mapping must be relinquished by calling

        boolean UnmapBufferARB(enum target);

    with <target> set to ARRAY_BUFFER_ARB.  Unmapping a mapped buffer
    object invalidates the pointers to its data store and sets the
    object's BUFFER_MAPPED_ARB state to FALSE and its
    BUFFER_MAP_POINTER_ARB state to NULL.

    UnmapBufferARB returns TRUE unless data values in the buffer's data
    store have become corrupted during the period that the buffer was
    mapped.  Such corruption can be the result of a screen resolution
    change or other window-system-dependent event that causes system
    heaps such as those for high-performance graphics memory to be
    discarded.  GL implementations must guarantee that such corruption
    can occur only during the periods that a buffer's data store is
```

mapped. If such corruption has occurred, UnmapBufferARB returns FALSE, and the contents of the buffer's data store become undefined.

It is an INVALID_OPERATION error to explicitly unmap a buffer data store that is in the unmapped state. Unmapping that occurs as a side effect of buffer deletion or reinitialization is not an error, however."

2.8A.1 Vertex Arrays in Buffer Objects

Blocks of vertex array data may be stored in buffer objects with the same format and layout options supported for client-side vertex arrays. However, it is expected that GL implementations will (at minimum) be optimized for data with all components represented as floats, as well as for color data with components represented as either floats or unsigned bytes.

A buffer object binding point is added to the client state associated with each vertex array type. The client does not directly specify the bindings to with these new binding points. Instead, the commands that specify the locations and organizations of vertex arrays copy the buffer object name that is bound to ARRAY_BUFFER_ARB to the binding point corresponding to the vertex array of the type being specified. For example, the NormalPointer command copies the value of ARRAY_BUFFER_BINDING_ARB (the queriable name of the buffer binding corresponding to the target ARRAY_BUFFER_ARB) to the client state variable NORMAL_ARRAY_BUFFER_BINDING_ARB.

If EXT_vertex_shader is defined, then the command VariantArrayEXT(uint id, ...) copies the value of ARRAY_BUFFER_BINDING_ARB to the buffer object binding point corresponding to variant array <id>.

If ARB_vertex_program is defined, then the command VertexAttribPointerARB(int attrib, ...) copies the value of ARRAY_BUFFER_BINDING_ARB to the buffer object binding point corresponding to vertex attrib array <attrib>.

If ARB_vertex_blend is defined, then the command WeightPointerARB copies the value of ARRAY_BUFFER_BINDING_ARB to WEIGHT_ARRAY_BUFFER_BINDING_ARB.

Rendering commands ArrayElement, DrawArrays, DrawElements, DrawRangeElements, MultiDrawArrays, and MultiDrawElements operate as previously defined, except that data for enabled vertex, variant, and attrib arrays are sourced from buffers if the array's buffer binding is non-zero. When an array is sourced from a buffer object, the pointer value of that array is used to compute an offset, in basic machine units, into the data store of the buffer object. This offset is computed by subtracting a null pointer from the pointer value, where both pointers are treated as pointers to basic machine units.

It is acceptable for vertex, variant, or attrib arrays to be sourced from any combination of client memory and various buffer objects during a single rendering operation.

It is an INVALID_OPERATION error to source data from a buffer object that is currently mapped.

2.8B.1 Array Indices in Buffer Objects
--

Blocks of array indices may be stored in buffer objects with the same format options that are supported for client-side index arrays. Initially zero is bound to ELEMENT_ARRAY_BUFFER_ARB, indicating that DrawElements and DrawRangeElements are to source their indices from arrays passed as their <indices> parameters, and that MultiDrawElements is to source its indices from the array of pointers to arrays passed in as its <indices> parameter.

A buffer object is bound to ELEMENT_ARRAY_BUFFER_ARB by calling

```
    void BindBufferARB(enum target, uint buffer);
```

with <target> set to ELEMENT_ARRAY_BUFFER_ARB, and <buffer> set to the name of the buffer object. If no corresponding buffer object exists, one is initialized as defined in Section 2.8A.

The commands BufferDataARB, BufferSubDataARB, MapBufferARB, and UnmapBufferARB may all be used with <target> set to ELEMENT_ARRAY_BUFFER_ARB. In such event, these commands operate in the same fashion as described in section 2.8A, but on the buffer currently bound to the ELEMENT_ARRAY_BUFFER_ARB target.

While a non-zero buffer object name is bound to ELEMENT_ARRAY_BUFFER_ARB, DrawElements and DrawRangeElements source their indices from that buffer object, using their <indices> parameters as offsets into the buffer object in the same fashion as described in section 2.8A1. MultiDrawElements also sources its indices from that buffer object, using its <indices> parameter as a pointer to an array of pointers that represent offsets into the buffer object.

Buffer objects created by binding an unused name to ARRAY_BUFFER_ARB and to ELEMENT_ARRAY_BUFFER_ARB are formally equivalent, but the GL may make different choices about storage implementation based on the initial binding. In some cases performance will be optimized by storing indices and array data in separate buffer objects, and by creating those buffer objects with the corresponding binding points."

Additions to Chapter 3 of the 1.4 Specification (Rasterization)

None

Additions to Chapter 4 of the 1.4 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.4 Specification (Special Functions)

Added to section 5.4, as part of the discussion of what commands are compiled into display lists:

"Commands that are used to create, manage, and query buffer objects are not included in display lists, but are executed immediately. These commands are BindBufferARB, DeleteBuffersARB, GenBuffersARB, IsBufferARB, BufferDataARB, BufferSubDataARB, MapBufferARB, UnmapBufferARB, GetBufferParameterivARB, GetBufferSubDataARB, and GetBufferPointervARB.

GL commands that source data from buffer objects dereference the buffer object data in question at display list compile time, rather than encoding the buffer ID and buffer offset into the display list. Only GL commands that are executed immediately, rather than being compiled into a display list, are permitted to use a buffer object as a data sink."

Additions to Chapter 6 of the 1.4 Specification (State and State Requests)

Added to section 6.1 in a subsection titled Buffer Object Queries:

"The command

```
    boolean IsBufferARB(uint buffer);
```

returns TRUE if <buffer> is the name of an buffer object. If <buffer> is zero, or if <buffer> is a non-zero value that is not the name of an buffer object, IsBufferARB return FALSE.

The command

```
    void GetBufferSubDataARB(enum target, intptrARB offset,
                             sizeiptrARB size, void *data);
```

queries the data contents of a buffer object. <target> is ARRAY_BUFFER_ARB. <offset> and <size> indicate the range of data in the buffer object that is to be queried, in terms of basic machine units. <data> specifies a region of client memory, <size> basic machine units in length, into which the data is to be retrieved.

An error is generated if GetBufferSubDataARB is executed for a buffer object that is currently mapped.

While the data store of a buffer object is mapped, the pointer to the data store can be queried by calling

```
    void GetBufferPointervARB(enum target, enum pname, void **params);
```

with <target> set to ARRAY_BUFFER_ARB and <pname> set to BUFFER_MAP_POINTER_ARB. The single buffer map pointer is returned in <params>. GetBufferPointervARB returns the NULL pointer value if the buffer's data store is not currently mapped, or if the requesting client did not map the buffer object's data store, and the implementation is unable to support mappings on multiple clients."

Added to the list of queries in section 6.1.3, Enumerated Queries:

    "void GetBufferParameterivARB(enum target, enum pname, int *params);"

Errors

INVALID_ENUM is generated if the <target> parameter of BindBufferARB, BufferDataARB, BufferSubDataARB, MapBufferARB, UnmapBufferARB, GetBufferSubDataARB, GetBufferParameterivARB, or GetBufferPointervARB is not ARRAY_BUFFER_ARB or ELEMENT_ARRAY_BUFFER_ARB.

INVALID_VALUE is generated if the <n> parameter of DeleteBuffersARB or GenBuffersARB is negative.

INVALID_VALUE is generated if the <size> parameter of BufferDataARB, BufferSubDataARB, or GetBufferSubDataARB is negative.

INVALID_OPERATION is generated if BufferDataARB, BufferSubDataARB, MapBufferARB, UnmapBufferARB, GetBufferSubDataARB, GetBufferParameterivARB, or GetBufferPointervARB is executed while zero is bound to the <target> parameter.

OUT_OF_MEMORY may be generated if the data store of a buffer object cannot be allocated because the <size> argument of BufferDataARB is too large.

OUT_OF_MEMORY may be generated when MapBufferARB is called if the data store of the buffer object in question cannot be mapped. This may occur for a variety of system-specific reasons, such as the absence of sufficient remaining virtual memory.

INVALID_ENUM is generated if the <usage> parameter of BufferDataARB is not STREAM_DRAW_ARB, STREAM_READ_ARB, STREAM_COPY_ARB, STATIC_DRAW_ARB, STATIC_READ_ARB, STATIC_COPY_ARB, DYNAMIC_DRAW_ARB, DYNAMIC_READ_ARB, or DYNAMIC_COPY_ARB.

INVALID_VALUE is generated if the <offset> parameter to BufferSubDataARB or GetBufferSubDataARB is negative.

INVALID_VALUE is generated if the <offset> and <size> parameters of BufferSubDataARB or GetBufferSubDataARB define a region of memory that extends beyond that allocated by BufferDataARB.

INVALID_OPERATION is generated if MapBufferARB is executed for a buffer that is already mapped.

INVALID_OPERATION is generated if UnmapBufferARB is executed for a buffer that is not currently mapped.

INVALID_ENUM is generated if the <access> parameter of MapBufferARB is not READ_ONLY_ARB, WRITE_ONLY_ARB, or READ_WRITE_ARB.

INVALID_ENUM is generated if the <pname> parameter of GetBufferParameterivARB is not BUFFER_SIZE_ARB, BUFFER_USAGE_ARB, BUFFER_ACCESS_ARB, or BUFFER_MAPPED_ARB.

INVALID_ENUM is generated if the <pname> parameter of GetBufferPointervARB is not BUFFER_MAP_POINTER_ARB.

INVALID_OPERATION may be generated if any of the commands defined in this extension is executed between the execution of Begin and the corresponding execution of End.

INVALID_OPERATION is generated if a buffer object that is currently mapped is used as a source of GL render data, or as a destination of GL query data.

INVALID_OPERATION is generated if BufferSubDataARB is used to modify the data store contents of a mapped buffer, or if GetBufferSubDataARB is used to query to data store contents of a mapped buffer.

New State

(table 6.7, Vertex Array Data, p. 222)

Get Value	Type	Get Command	Initial Value	Sec	Attribute
ARRAY_BUFFER_BINDING_ARB	Z+	GetIntegerv	0	2.8A	vertex-array
VERTEX_ARRAY_BUFFER_BINDING_ARB	Z+	GetIntegerv	0	2.8A	vertex-array
NORMAL_ARRAY_BUFFER_BINDING_ARB	Z+	GetIntegerv	0	2.8A	vertex-array
COLOR_ARRAY_BUFFER_BINDING_ARB	Z+	GetIntegerv	0	2.8A	vertex-array
INDEX_ARRAY_BUFFER_BINDING_ARB	Z+	GetIntegerv	0	2.8A	vertex-array
TEXTURE_COORD_ARRAY_BUFFER_BINDING_ARB	Z+	GetIntegerv	0	2.8A	vertex-array
EDGE_FLAG_ARRAY_BUFFER_BINDING_ARB	Z+	GetIntegerv	0	2.8A	vertex-array
SECONDARY_COLOR_ARRAY_BUFFER_BINDING_ARB	Z+	GetIntegerv	0	2.8A	vertex-array
FOG_COORDINATE_ARRAY_BUFFER_BINDING_ARB	Z+	GetIntegerv	0	2.8A	vertex-array
WEIGHT_ARRAY_BUFFER_BINDING_ARB	Z+	GetIntegerv	0	2.8A	vertex-array
ELEMENT_ARRAY_BUFFER_BINDING_ARB	Z+	GetIntegerv	0	2.8A.2	vertex-array
VERTEX_ATTRIB_ARRAY_BUFFER_BINDING_ARB	16+ x Z+	GetVertexAttribivARB	0	2.8A	vertex-array

XXX need to add buffer state for variant arrays

(new table for buffer objects)

Get Value	Type	Get Command	Initial Value	Sec	Attribute
(buffer data)	BMU	GetBufferSubDataARB		2.8A	none
BUFFER_SIZE_ARB	Z+	GetBufferParameterivARB	0	2.8A	none
BUFFER_USAGE_ARB	Z9	GetBufferParameterivARB	STATIC_DRAW_ARB	2.8A	none
BUFFER_ACCESS_ARB	Z3	GetBufferParameterivARB	READ_WRITE_ARB	2.8A	none
BUFFER_MAPPED_ARB	B	GetBufferParameterivARB	FALSE	2.8A	none
BUFFER_MAP_POINTER_ARB	Y	GetBufferPointervARB	NULL	2.8A	none

New Implementation Dependent State

(none)

Usage Examples

These examples illustrate various usages. In all cases a rendering loop is included, and array parameters are initialized inside the loop as would be required if multiple array rendering operations were performed in the loops. (Though only one operation is shown.)

Convenient macro definition for specifying buffer offsets:

```
    #define BUFFER_OFFSET(i) ((char *)NULL + (i))
```

Traditional vertex arrays:

```
    // Create system memory buffer
    data = malloc(320);

    // Fill system memory buffer
    ...

    // Frame rendering loop
    while (...) {

        // Define arrays
        VertexPointer(4, FLOAT, 0, data);
        ColorPointer(4, UNSIGNED_BYTE, 0, data+256);

        // Enable arrays
        EnableClientState(VERTEX_ARRAY);
        EnableClientState(COLOR_ARRAY);

        // Draw arrays
        DrawArrays(TRIANGLE_STRIP, 0, 16);

        // Disable arrays
        DisableClientState(VERTEX_ARRAY);
        DisableClientState(COLOR_ARRAY);

        // Other rendering commands
        ...

    }

    // Free system memory buffer
    free(data);
```

Vertex arrays using a buffer object:

```
    // Create system memory buffer
    data = malloc(320);

    // Fill system memory buffer
    ...

    // Create buffer object
    BindBufferARB(ARRAY_BUFFER_ARB, 1);

    // Initialize data store of buffer object
    BufferDataARB(ARRAY_BUFFER_ARB, 320, data, STATIC_DRAW_ARB);

    // Free system memory buffer
    free(data);

    // Frame rendering loop
    while (...) {

        // Define arrays
        BindBufferARB(ARRAY_BUFFER_ARB, 1);
        VertexPointer(4, FLOAT, 0, BUFFER_OFFSET(0));
        ColorPointer(4, UNSIGNED_BYTE, 0, BUFFER_OFFSET(256));

        // Enable arrays
        EnableClientState(VERTEX_ARRAY);
        EnableClientState(COLOR_ARRAY);

        // Draw arrays
        DrawArrays(TRIANGLE_STRIP, 0, 16);

        // Disable arrays
        DisableClientState(VERTEX_ARRAY);
        DisableClientState(COLOR_ARRAY);

        // Other rendering commands
        ...

    }

    // Delete buffer object
    int buffer[1] = {1};
    DeleteBuffersARB(1, buffer);
```

Code that works with and without buffer objects:

```
        // Create system memory buffer
        data = malloc(320);

        // Fill system memory buffer
        ...

        // Initialize buffer object, and null the data pointer
#ifdef USE_BUFFER_OBJECTS
        BindBufferARB(ARRAY_BUFFER_ARB, 1);
        BufferDataARB(ARRAY_BUFFER_ARB, 320, data, STATIC_DRAW_ARB);
        free(data);
        data = NULL;
#endif

        // Frame rendering loop
        while (...) {

            // Define arrays
#ifdef USE_BUFFER_OBJECTS
            BindBufferARB(ARRAY_BUFFER_ARB, 1);
#endif
            VertexPointer(4, FLOAT, 0, data);
            ColorPointer(4, UNSIGNED_BYTE, 0, data+256);

            // Enable arrays
            EnableClientState(VERTEX_ARRAY);
            EnableClientState(COLOR_ARRAY);

            // Draw arrays
            DrawArrays(TRIANGLE_STRIP, 0, 16);

            // Disable arrays
            DisableClientState(VERTEX_ARRAY);
            DisableClientState(COLOR_ARRAY);

            // Other rendering commands
            ...

        }

        // Delete buffer object
#ifdef USE_BUFFER_OBJECTS
        int buffer[1] = {1};
        DeleteBuffersARB(1, buffer);
#else
        // Free system memory buffer
        free(data);
#endif
```

Vertex arrays using a mapped buffer object:

```
    // Frame rendering loop
    while (...) {

        // Define arrays (and create buffer object in first pass)
        BindBufferARB(ARRAY_BUFFER_ARB, 1);
        VertexPointer(4, FLOAT, 0, BUFFER_OFFSET(0));
        ColorPointer(4, UNSIGNED_BYTE, 0, BUFFER_OFFSET(256));

        // Enable arrays
        EnableClientState(VERTEX_ARRAY);
        EnableClientState(COLOR_ARRAY);

        // Initialize data store of buffer object
        BufferDataARB(ARRAY_BUFFER_ARB, 320, NULL, STREAM_DRAW_ARB);

        // Map the buffer object
        float *p = MapBufferARB(ARRAY_BUFFER_ARB, WRITE_ONLY);

        // Compute and store data in mapped buffer object
        ...

        // Unmap buffer object and draw arrays
        if (UnmapBufferARB(ARRAY_BUFFER_ARB)) {
            DrawArrays(TRIANGLE_STRIP, 0, 16);
        }

        // Disable arrays
        DisableClientState(VERTEX_ARRAY);
        DisableClientState(COLOR_ARRAY);

        // Other rendering commands
        ...

    }

    // Delete buffer object
    int buffer[1] = {1};
    DeleteBuffersARB(1, buffer);
```

Vertex arrays using a mapped buffer object for array data and an unmapped buffer object for indices:

```
    // Create system memory buffer for indices
    indexdata = malloc(400);

    // Fill system memory buffer with 100 indices
    ...

    // Create index buffer object
    BindBufferARB(ELEMENT_ARRAY_BUFFER_ARB, 2);
    BufferDataARB(ELEMENT_ARRAY_BUFFER_ARB, 400, indexdata,
            STATIC_DRAW_ARB);

    // Free system memory buffer
    free(indexdata);

    // Frame rendering loop
    while (...) {

        // Define arrays (and create buffer object in first pass)
        BindBufferARB(ARRAY_BUFFER_ARB, 1);
        VertexPointer(4, FLOAT, 0, BUFFER_OFFSET(0));
        ColorPointer(4, UNSIGNED_BYTE, 0, BUFFER_OFFSET(256));
        BindBufferARB(ELEMENT_ARRAY_BUFFER_ARB, 2);

        // Enable arrays
        EnableClientState(VERTEX_ARRAY);
        EnableClientState(COLOR_ARRAY);

        // Initialize data store of buffer object
        BufferDataARB(ARRAY_BUFFER_ARB, 320, NULL, STREAM_DRAW_ARB);

        // Map the buffer object
        float *p = MapBufferARB(ARRAY_BUFFER_ARB, WRITE_ONLY);

        // Compute and store data in mapped buffer object
        ...

        // Unmap buffer object and draw arrays
        if (UnmapBufferARB(ARRAY_BUFFER_ARB)) {
            DrawElements(TRIANGLE_STRIP, 100, UNSIGNED_INT,
                         BUFFER_OFFSET(0));
        }

        // Disable arrays
        DisableClientState(VERTEX_ARRAY);
        DisableClientState(COLOR_ARRAY);

        // Other rendering commands
        ...

    }

    // Delete buffer objects
    int buffers[2] = {1, 2};
    DeleteBuffersARB(1, buffers);
```

Mapping multiple buffers simultaneously:

```
    // Map buffers
    BindBuffer(ARRAY_BUFFER_ARB, 1);
    float *a = MapBuffer(ARRAY_BUFFER_ARB, WRITE_ONLY);
    BindBuffer(ARRAY_BUFFER_ARB, 2);
    float *b = MapBuffer(ARRAY_BUFFER_ARB, WRITE_ONLY);

    // Fill buffers
    ...

    // Unmap buffers
    BindBuffer(ARRAY_BUFFER_ARB, 1);
    if (!UnmapBufferARB(ARRAY_BUFFER_ARB)) {
        // Handle error case
    }
    BindBuffer(ARRAY_BUFFER_ARB, 2);
    if (!UnmapBufferARB(ARRAY_BUFFER_ARB)) {
        // Handle error case
    }
```

Name

    ARB_vertex_program

Name Strings

    GL_ARB_vertex_program

Contributors

    Kurt Akeley
    Allen Akin
    Ben Ashbaugh
    Bob Beretta
    John Carmack
    Matt Craighead
    Ken Dyke
    Steve Glanville
    Michael Gold
    Evan Hart
    Mark Kilgard
    Bill Licea-Kane
    Barthold Lichtenbelt
    Erik Lindholm
    Benj Lipchak
    Bill Mark
    James McCombe
    Jeremy Morris
    Brian Paul
    Bimal Poddar
    Thomas Roell
    Jeremy Sandmel
    Jon Paul Schelter
    Geoff Stahl
    John Stauffer
    Nick Triantos

IP Status

NVIDIA claims to own intellectual property related to this extension, and has signed an ARB Contributor License agreement licensing this intellectual property.

Microsoft claims to own intellectual property related to this extension.

Status

Complete. Approved by ARB on June 18, 2002

Version

```
Last Modified Date:  07/25/07
Revision:            46
```

Number

ARB Extension #26

Dependencies

Written based on the wording of the OpenGL 1.3 specification and requires OpenGL 1.3.

ARB_vertex_blend and EXT_vertex_weighting affect the definition of this extension.

ARB_matrix_palette affects the definition of this extension.

ARB_point_parameters and EXT_point_parameters affect the definition of this extension.

EXT_secondary_color affects the definition of this extension.

EXT_fog_coord affects the definition of this extension.

ARB_transpose_matrix affects the definition of this extension.

NV_vertex_program interacts with this extension.

EXT_vertex_shader interacts with this extension.

Overview

Unextended OpenGL mandates a certain set of configurable per-vertex computations defining vertex transformation, texture coordinate generation and transformation, and lighting. Several extensions have added further per-vertex computations to OpenGL. For example, extensions have defined new texture coordinate generation modes (ARB_texture_cube_map, NV_texgen_reflection, NV_texgen_emboss), new vertex transformation modes (ARB_vertex_blend, EXT_vertex_weighting), new lighting modes (OpenGL 1.2's separate specular and rescale normal functionality), several modes for fog distance generation (NV_fog_distance), and eye-distance point size attenuation (EXT/ARB_point_parameters).

Each such extension adds a small set of relatively inflexible per-vertex computations.

This inflexibility is in contrast to the typical flexibility provided by the underlying programmable floating point engines (whether micro-coded vertex engines, DSPs, or CPUs) that are traditionally used to implement OpenGL's per-vertex computations. The purpose of this extension is to expose to the OpenGL application writer a significant degree of per-vertex programmability for computing vertex parameters.

For the purposes of discussing this extension, a vertex program is a sequence of floating-point 4-component vector operations that determines how a set of program parameters (defined outside of OpenGL's Begin/End pair) and an input set of per-vertex parameters are transformed to a set of per-vertex result parameters.

The per-vertex computations for standard OpenGL given a particular set of lighting and texture coordinate generation modes (along with any state for extensions defining per-vertex computations) is, in essence, a vertex program. However, the sequence of operations is defined implicitly by the current OpenGL state settings rather than defined explicitly as a sequence of instructions.

This extension provides an explicit mechanism for defining vertex program instruction sequences for application-defined vertex programs. In order to define such vertex programs, this extension defines a vertex programming model including a floating-point 4-component vector instruction set and a relatively large set of floating-point 4-component registers.

The extension's vertex programming model is designed for efficient hardware implementation and to support a wide variety of vertex programs. By design, the entire set of existing vertex programs defined by existing OpenGL per-vertex computation extensions can be implemented using the extension's vertex programming model.

Issues

(1) What should this extension be called?

RESOLVED: ARB_vertex_program. DirectX 8 refers to its similar functionality as "vertex shaders". This is a confusing term because shaders are usually assumed to operate at the fragment or pixel level, not the vertex level.

Conceptually, what the extension defines is an application-defined program (admittedly limited by its sequential execution model) for processing vertices so the "vertex program" term is more accurate.

Some of the API machinery in this extension for describing programs should be useful for extending other OpenGL operations with programs (though other types of programs may look very different from vertex programs).

(2) What terms are important to this specification?

vertex program mode - When vertex program mode is enabled, vertices are transformed by an application-defined vertex program.

conventional GL vertex transform mode - When vertex program mode is disabled (or the extension is not supported), vertices are transformed by GL's conventional texgen, lighting, and transform state.

vertex program - An application-defined program used to transform vertices when vertex program mode is enabled.

program target - A type or class of program. This extension supports the VERTEX_PROGRAM_ARB target. Future extensions may add other program targets.

program object - An object maintained internal to OpenGL that encapsulates a program and a set of associated state. Operations performed on program objects include loading a program, binding, generating program object names, querying state, and deleting.

program name - Each program object has an associated unsigned integer, called the program name. Applications refer to a program object using the program name.

current program - Each program target may have a current program object. For vertex programs, the current program is executed whenever a vertex is specified when vertex program mode is enabled.

default program - Each program target has a default program object, referred to using a program name of zero. The current program for each program target is initially the default program for that target.

program execution environment - A set of resources, instructions, and semantic rules used to execute a program. Each program target may support one or more execution environment -- new execution environments may provide new instructions, resources, or execution rules. Program strings must specify the execution environment that should be used to execute the program.

program options - An optional feature that modifies the rules of the execution environment. Vertex programs specify the options that they require at the beginning of the program.

vertex attribute - GL state associated with vertices that can vary per vertex.

conventional vertex attributes - Per-vertex attributes used in conventional GL vertex transform mode, including colors, normals, texture coordinate sets.

generic vertex attributes - An array of 16+ 4-component vectors added by this extension. Generic vertex attributes can be used by vertex programs but are unused in conventional GL vertex transform mode.

program parameter - A set of constants that are available for vertex programs to use during their execution. Program parameters include program environment parameters, program local parameters, conventional GL state, and program constants.

program environment parameter - A set of 96+ 4-component vectors belonging to the GL context that can be used as constants during the execution of any vertex program.

program local parameter - A set of 96+ 4-component vectors belonging to a vertex program object that can be used as constants during the execution of the corresponding vertex program. Program local parameters can not be used by any other vertex programs.

program constants - Constants declared in the text of a program may be used during the execution of that program.

program temporaries - A set of 12+ 4-component vectors to hold temporary results that can be read or written during the execution of a vertex program.

program address registers - A set of 1+ 1-component integer vectors that can be used to perform variable indirect accesses to program parameter arrays during the execution of a vertex program. Address registers are specified as vectors to allow for future extensions supporting multiple address register components.

program results - A set of 4-component vectors to hold the final results of a vertex program. The program results correspond closely to the set of vertex attributes used during primitive assembly and rasterization.

program variables - Variable names used to identify a specific vertex attribute, program parameter, temporary, address register, or result.

program binding - A program statement that declares a variable and associates it with a specific vertex attribute, program parameter, or program result.

implicit binding - When an executable instruction refers to a specific vertex attribute, program parameter, program result, or constant by name, without using an explicit program binding statement. When such values are encountered, an implicit binding to an anonymous variable name is created.

program invocation - The act of implicitly or explicitly kicking off program execution. Vertex programs are invoked automatically when vertex program mode is enabled and vertices are received. Vertex programs are also invoked automatically when the current raster position is specified.

(3) What part of OpenGL do vertex programs specifically bypass?

Vertex programs bypass the following OpenGL functionality:

- The modelview and projection matrix vertex transformations.

- Vertex weighting/blending (ARB_vertex_blend).

- Normal transformation, rescaling, and normalization.

- Color material.

- Per-vertex lighting.

- Texture coordinate generation and texture matrix transformations.

- Per-vertex point size computations in ARB/EXT_point_parameters

- Per-vertex fog coordinate computations in EXT_fog_coord and NV_fog_distance.

- Client-defined clip planes.

- The normalization of AUTO_NORMAL evaluated normals

- All of the above, when computing the current raster position.

Operations not subsumed by vertex programs

- Clipping to the view frustum.

- Perspective divide (division by w).

- The viewport transformation.

- The depth range transformation.

- Front and back color selection (for two-sided lighting and coloring).

- Clamping the primary and secondary colors to [0,1].

- Primitive assembly and subsequent operations.

- Evaluators (except the AUTO_NORMAL normalization).

(5) This extension adds a set of generic vertex attributes to the existing conventional attributes. The sum of the number of generic and conventional attributes supported on a given platform may exceed the total number of per-vertex attributes supported in hardware. How should this situation be handled?

RESOLVED: Implementations may alias conventional and generic vertex attributes, where pairs of conventional and generic vertex attributes share the same storage. Such aliasing will effectively reduce the number of vertex attributes a hardware platforms. While implementations may alias attributes, that behavior is not required. To accommodate both behaviors, changing a generic vertex attribute leaves the corresponding conventional attribute undefined, and vice versa.

This undefined behavior is a compromise between the existing EXT_vertex_shader extension (which does not permit aliasing) and the NV_vertex_program extension (which requires aliasing). The mapping between generic and conventional vertex attributes is found in Table X.1 below. This mapping is taken from the NV_vertex_program specification and generalized to define behavior for >8 texture coordinate sets.

Applications that mix conventional and generic vertex attributes in a single rendering pass should be careful to avoid using attributes that may alias. To limit inadvertent use of such attributes, loading a vertex program that used a pair of attributes that may alias is guaranteed to fail. Applications requiring a small number of generic vertex attributes can always safely use generic attributes 6 and 7, and any supported attributes corresponding to unused or unsupported texture units. For example, if an implementation supports only four texture units, generic attributes 12 through 15 can always be used safely.

(6) Should there be a "VertexAttribs" entry point to specify multiple vertex attributes in one immediate mode call.

RESOLVED: No. Not providing such functionality serves to reduce the already large number of required immediate mode entry points. A "VertexAttribs" command would improve the efficiency of vertex attribute transfer, but vertex arrays or display lists should still be better.

(7) Should a full complement of data types (signed and unsigned bytes, shorts, and ints, as well as floats and doubles) be supported for vertex attributes? Should fixed-point data types be supported in both normalized (map the range to [0,1] or [-1,1]) and unnormalized form?

RESOLVED: For vertex arrays, all data type combinations are supported.

For immediate mode, a smaller subset is supported, to limit the number of immediate-mode entry points added by this extension. In fully general form, 112 immediate-mode entry points (4 sizes x 2 vector/non-vector x 14 data types) would be required.

Immediate mode support is available for non-normalized shorts, floats, and doubles for all component counts. Additionally, immediate mode support is available for 4-component vectors of all data types (normalized and unnormalized).

Note also that in immediate mode, the "N" qualifier in function names like VertexAttrib4Nub will be used to indicate that fixed-point data should be normalized.

(8) How should applications indicate that fixed-point generic vertex attribute array components should be converted to [-1,+1] or [0,1] ranges?

RESOLVED: The function VertexAttribPointerARB takes a boolean argument <normalized> that indicates whether fixed-point array data should be normalized to [-1,+1] or [0,1].

One alternate approach would have been to extend to set of enumerants to include values such as NORMALIZED_UNSIGNED_BYTE_ARB. Adding such enumerants in some sense implies that UNSIGNED_BYTE is not normalized, even though it usually is.

(9) In unextended OpenGL, calling Vertex() specifies a vertex and causes vertex transformation operations to be performed on the vertex. Should there be an equivalent method to specify a vertex using generic vertex attributes? If so, how should this be accomplished?

RESOLVED: Setting generic vertex attribute zero will always specify a vertex. Vertex*(...) and VertexAttrib*(0,...) are specified to be equivalent, whether or not vertex program mode is enabled. Allowing generic vertex attribute zero to specify a vertex allows applications to write vertex programs that use only generic attributes; otherwise, applications would have had to use Vertex() to provoke vertex processing.

(10) How is this extension different from previous vertex program extensions, such as EXT_vertex_shader or NV_vertex_program? What pitfalls are there in porting vertex programs to/from this extension?

RESOLVED: See "Interactions with NV_vertex_program" and "Interactions with EXT_vertex_shader" sections near the end of this specification.

(11) Should program parameter variables bound to GL state be updated automatically after the bound state changes? If so, when?

RESOLVED: Yes. Such variables are updated automatically prior to the next vertex program invocation with no application intervention required. A proposal to reduce the burden by requiring a manual "update state" step was considered and rejected.

(12) How should this specification handle variable bindings to Material state? Material is allowed inside a Begin/End, so material properties are technically per-vertex state.

RESOLVED: Materials can be bound only as program parameters. Changes to material properties inside a Begin/End will leave the bindings undefined until the subsequent End command. At that point, all material property bindings are guaranteed to be updated, and any material property changes up to the next Begin command are guaranteed to take effect immediately.

Supporting per-vertex material properties places additional pressure on the number of per-vertex bindings an implementation can support, which was already a problem. See issue (5).

In practice, material properties are usually not changed in this manner. Applications needing to change material properties inside a Begin/End in vertex program mode can work around this limitation by storing the color in a conventional or generic vertex attribute and modifying the vertex program accordingly.

(13) What semantic restrictions, if any, should be imposed on binding the same GL state to multiple variables? The grammar permits such bindings, but allowing this behavior means that single state updates must update multiple variables.

RESOLVED: Cases where a single state update necessarily requires updating multiple variables are disallowed. The only restriction resulting from this decision is that a single state variable can not be bound more than once in the collection of arrays that are accessed using relative addressing (at run time). The driver can and will coalesce all other bindings accessed only at fixed offsets into a single binding.

This restriction and a little driver work allows the same state variable to be used multiple times without requiring that a single state change update multiple variables.

(14) What semantic restrictions, if any, should be imposed on using multiple vertex attributes or program parameters in the same instruction?

RESOLVED: None. If the underlying hardware implementation does not support reads of multiple attributes or program parameters, the driver may need to transparently insert additional instructions and/or consume temporaries to perform the operation.

(15) How and when should related state be combined into a single program parameter binding? Additionally, should any values derived from core GL state be exposed, too?

RESOLVED: Related state should be combined when possible, as long as the binding name remains somewhat sensible. Additionally, certain pre-computed state items useful for performance reasons are also exposed. In particular, the following GL state combinations are supported:

* Light attenuation constants and the spot light exponent are combined into a single vector called "state.light[n].attenuation" (spot lights can attenuate the lit result).

* Spot light direction and cutoff angle cosine are called "state.light[n].spot.direction" (cutoff is directional information). Binding the cutoff angle itself is pretty useless, so the cosine is used.

* A pre-computed half angle for lighting with infinite lights and an infinite viewer is provided and called "state.light[n].half".

* Pre-computed products of ambient, diffuse, and specular light colors with the corresponding front or back material colors are supported, and are called "state.lightprod[n].<face>.<property>".

* Exponential fog density, linear fog start and end parameters, as well as the pre-computed reciprocal of (end-start) are combined into one vector called "state.fog.params".

* The core point size, minimum and maximum size clamps (ARB_point_parameters), and multisample fade size threshold (ARB_point_parameters) are combined into a single vector called "state.point.size".

* Precomputed transpose, inverse, and inverse transpose matrices are supported for each base matrix type.

(16) Should the initial values of temporaries and results be undefined?

RESOLVED: Since the underlying hardware differs, it was decided to leave these values uninitalized. There are a few issues related to this behavior that programs should keep in mind:

* Since any results not written by the program are undefined, programs should write to all result registers that are needed during rasterization.

* In particular, the initial vertex position result is undefined, and will remain undefined if not written by a program. To achieve proper results, vertex programs should be careful to write to all four components of the vertex position. Otherwise, primitives may be completely clipped or produce undefined results during rasterization. There is no semantic requirement that programs must write a transformed vertex position, so erroneous programs will load succesfully, but will produce undefined (and probably useless) results. Such a semantic requirement may be impossible to enforce in future language versions that support run-time branching.

* Since vertex programs may be executed when the raster position is set, any attributes not written by the program will result in undefined state in the current raster position. Programs should write to all result registers that would be used when rasterizing pixel primitives using the current raster position.

* If conventional OpenGL texture mapping operations are performed, a program should always write to the "w" coordinate of any texture coordinates result registers it needs to use. Conventional OpenGL texture accesses always use projective texture coordinates (e.g., s/q, t/q, r/q), even though q is almost always 1.0. An undefined q coordinate (coming from the "w" component of the result register) may produce undefined coordinates on the texture lookup.

(17) Should vertex programs be required to have a header token and an end token?

RESOLVED: Yes. The header token for this extension is named "!!ARBvp1.0". The ARB may standardize future language versions which would be expected to have tokens like "!!ARBvp2.0". Vertex programs must end with the "END" token.

The initial header token reminds the programmer what type of program they are writing. If vertex programs are ever read from disk files, the header token can be used to specifically identify vertex programs. The initial header tokens will also make it easier for programmers to distinguish between multiple types of vertex programs and between vertex programs and another future type of programs.

We expect that programs may be generated by concatenation of program fragments. The "END" token will hopefully reduce bugs due to specifying an incorrectly concatenated program.

(18) Should ProgramStringARB take a <program> specifier? Should ProgramLocalParameterARB and GetProgramLocalParameterARB take a <program> specifier? How about GetProgramivARB and GetProgramStringARB?

RESOLVED: No to all. Instead, these calls are specified to always query or modify the currently bound program object. Using bound objects allows GL implementations to avoid locking and name lookup overhead on each such call.

This behavior does imply that applications loading a sequence of program objects must bind each in turn.

(19) Should relative addressing be performed using an address register (load up an integer register) or by taking a floating-point scalar?

RESOLVED: Address register. It would not be a good idea to support both syntaxes simultaneously, since using a floating-point scalar may consume the only available address register in the process. The current address register syntax can be easily extended to allow for multiple integer registers and/or enable other integer operations in a later extension.

Using a floating-point index may require an extra instruction on some architectures, and would require optimization work to eliminate redundant loads. Using a floating-point index may consume one of a small number of temporary registers. On the other hand, for implementations without a dedicated address register, it may be necessary to dedicate a general-purpose register (or register component) to hold the address register contents.

(20) How should user-defined clipping be supported in this specification?

RESOLVED: User-defined clipping is not supported in standard vertex program mode. User-defined clipping support will be provided for programs that use the "position invariant" option, where all vertex transformation operations are performed by the fixed-function pipeline.

It is expected that future vertex program extensions or a future language standard may provide more powerful user clipping functionality.

The options considered were:

(1) Not at all. Does not work for applications requiring user clipping. User clipping could be supported through a language extension.

(2) Support only through the "position_invariant" option, where vertex transformation is performed by the fixed-function pipeline.

(3) Support by using the fixed-function pipeline to generate eye coordinates and perform user clipping as specified for conventional transformation. May not work properly if the vertex transformation doesn't match the standard "multiply by modelview and projection matrices" model.

(4) Project existing fixed-function clip planes into clip coordinates and perform the clip test in clip space. The clip planes would be transformed by the inverse of the projection matrix, which will not work if the projection matrix is singular.

(5) Provide a 4-component "user clip coordinate" result that can be bound by a vertex program. User clipping is performed as in unextended OpenGL, using the "user clip coordinate" in place of the non-existant eye coordinates. This approach allows an application to do user clipping in any coordinate system. Clipping would not be independent of primitive tesselation as in the conventional pipeline. Additionally, the implicit transformation of specified clip planes by the modelview matrix may be undesirable (e.g., clipping in object coordinates).

(6) Provide one or more "clip plane distance" results that can be bound by a vertex program. For conventional clipping applications, vertex programs would compute the dot products normally computed by fixed-function hardware. Additionally, this method would enable additional unconventional clipping effects. Primitives would be clipped to the portion whose interpolated clip distances are greater than or equal to zero. This approach has the same issues as (5).

(21) How should future vertex program opcodes be named?

RESOLVED: Three-character names are recommended for brevity. Three character names are not a hard-and-fast requirement; extra characters may be needed for clarity or to disambiguate instructions.

(22) Should anything be said about the precision used for carrying out the instructions?

RESOLVED: Not much; precision will vary across platforms. The minimum precision requirements (1 part in 10^5 or roughly 17 bits) are spelled out in section 2.1.1. In practice, implementations will generally provide precision comparable to that obtained using single precision floats. Documenting exact precision across implementations is difficult. Additionally, it is difficult to spell out precision requirements for "compound" operations such as DP4.

(23) Should this extension support evaluator maps for generic vertex attributes? If so, what attribute sizes should be supported? Note that such maps are not supported at all for texture units except zero.

RESOLVED: No. Evaluator support has not been consistently extended in previous extensions. For example, neither ARB_multitexture nor OpenGL 1.3 provide support for evaluators for texture units other than unit zero. Adding evaluators for generic attributes involves a large amount of new state and complexity, particularly if evaluators should be supported in general form (1, 2, 3, and 4 components, all supported data type).

(25) The number of generic vertex attributes is implementation-dependent and is at least 16. Each generic vertex attribute has a vertex array enable. Should two new entry points be provided to accept an arbitrary attribute number, or should we reserve a range of enumerants that is "large enough"?

RESOLVED: Yes. EnableVertexAttribArrayARB and DisableVertexAttribArrayARB. This allows the number of vertex attributes to be unbounded, instead of using a limited range.

(26) What limits should be imposed on the constants that can be added to or subtracted from the address register for relative addressing? Negative offsets are sometimes useful for shifting down in an array.

RESOLVED: -64 to +63 should be sufficient for the time being. Offset sizes are limited to allow offsets to be baked into device-dependent instruction encodings.

(28) What level of precision should be guaranteed for the EXP and LOG instructions? And for the EX2 and LG2 instructions?

RESOLVED: The low precision EXP and LOG instructions should give at least 10 bits (2^-11 maximum relative error). No specific treatment will be added for EX2/LG2, implying that the computations should at least meet the minimal floating-point precision required by the spec.

(29) Should incremental program compilation be supported?

RESOLVED: No. Applications can compile programs just as easily using string concatenation.

(30) Should the execution environment be identified by the program itself or as an additional "language" parameter to ProgramStringARB?

RESOLVED: Programs should identify their execution environment in the header. The header (plus any specified options) make it clear what kind of program is being defined.

(31) Should this extension provide support for character sets other than 7-bit ASCII?

RESOLVED: Provide a <format> argument to ProgramStringARB to allow for future extensions. Only ASCII will be supported by this extension; additional character sets or encodings could be supported using separate extensions.

(32) Support for "program object" functionality may be applicable to future program targets. Should this functionality be factored out into a separate extension?

RESOLVED: No, such separation is not necessary. This extension was designed to allow to easily accommodate future program target types. It would be straightforward to put program object functionality into a separate extension, but the functionality provided by that extension would be of no value by itself.

(33) Should program residency management be supported?

RESOLVED: No. This functionality can be supported in a separate extension if desired. If may be desirable to address residency management in a more general form, where an application may desire a diverse set of objects (textures, programs) to be resident at once.

(34) Should program object management APIs (GenProgramsARB, DeleteProgramsARB) work like texture objects or display lists?

RESOLVED: Texture objects.

Both approaches have their merits. Pluses for the display list model include: no need to keep around multiple indices if you want to allocate a group of object, contiguous indices may fall out on implementations that share one block allocator for textures and display lists. Pluses for the texture object model: non-contiguous indices may be more optimizable -- new objects can be mapped to empty blocks in a hash table to avoid collisions with existing objects, separate indices are more compatible with a future handle-based object paradigm, and a larger base of extensions using this model. Note that display list allocations needed to be contiguous to support CallLists, but no such requirement for texture or program objects exists for programs.

(35) Should there be support for a program object zero? With texture objects, texture object zero is "special" because it is the default texture object for each target type. Is there something like this for program objects?

RESOLVED: Yes. Like texture objects, there should be a separate program object zero for each program type. This allows applications to use vertex programs without needing to generate and manage program objects.

With texture objects, an object zero was needed for backward compatibility with pre-OpenGL 1.1 applications. There is no such requirement here, but providing an object zero nicely matches the familiar texture object model.

(36) How should this extension provide feedback on why a program failed to load?

RESOLVED: Two queries are provided. Calling GetIntegerv() with PROGRAM_ERROR_POSITION_ARB provides the offset of an offending instruction in the program string. An error position of -1 indicates that a program loaded successfully. Calling GetString() with PROGRAM_ERROR_STRING_ARB returns an implementation-dependent error string explaining the reason for the failure. The error string can be queried even on successful program loads to check for warning messages.

The error string may be kept in a static buffer managed by the GL implementation. Implementations may reuse the same buffer on subsequent calls to ProgramStringARB, so returned error strings are guaranteed to be valid only until the next such call.

(37) How does ARB_vertex_blend's WEIGHT_SUM_UNITY_ARB mode interact with this extension? This mode allows an application to specify N-1 weights, and have the Nth weight computed by the GL.

RESOLVED: The ARB_vertex_blend spec (as of May, 2002) specifies that the nth weight is automatically computed by the GL and is effectively current state. In practice, ARB_vertex_blend implementations compute the nth weight on the fly in the fixed-function transformation pipeline, implying that the ARB_vertex_blend spec may require a fix. For the purposes of this extension, the WEIGHT_SUM_UNITY_ARB enable is ignored in vertex program mode. Applications performing a vertex weighting operation in a vertex program are free to compute the extra weight in the program.

(38) Should program environment parameters be pushed and popped?

RESOLVED: No. There is no need to push and pop this large set of state, much like pixel maps. Adding a new attribute bit would have complicated logistics (would the bit be included in ALL_ATTRIB_BITS?). Having program local parameters provides a method for making localized changes to certain state simply by switching programs.

(39) How should this extension interact with color material?

RESOLVED: When color material is enabled, any bindings of material colors that track the current color should be updated when the current color is updated. In this specification, material properties can be bound only as program parameters, and any changes to the material properties inside a Begin/End leave the bindings undefined until the next End command. Similarly, any indirect changes to the material properties (through ColorMaterial) will have a similar effect.

Several other options were considered here. One option was to support per-vertex material property bindings and have programs that reference tracked material properties should get the current color. This could be handled either by broadcasting the current color to multiple vertex attributes, or recompiling the vertex program so that references to a tracked material property are redirected to the vertex color. Both such solutions are somewhat complex. A second option would be to ignore the COLOR_MATERIAL enable and instead use an "old" material color. This breaks the standard color material model. Implementations can and often do defer such updates (making an "old" color available), some conditions may cause an implementation to update of material state at odd times.

(41) What about when the execution environment involves support for other extensions? In particular, the execution environment subsumes some functionality from EXT/ARB_point_parameters, EXT_fog_coord, EXT_secondary_color, and ARB_multitexture.

RESOLVED: This extension assumes support for functionality that includes a fog coordinate, secondary color, per-vertex point sizes, and multiple texture coordinates (at least to the extent that it exposes >1 texture coordinate). All of these extensions are supported fairly widely. On some platforms, some of this functionality may require software fallbacks.

(42) How does PointSize work with vertex programs?

RESOLVED: If VERTEX_PROGRAM_POINT_SIZE_ARB is disabled, the size of points is determined by the PointSize state and is not attenuated, even if EXT_point_parameters is supported. If enabled, the point size is the point size result value, and is clamped to implementation-dependent point size limits during point rasterization.

(43) What do we say about the alpha component of the secondary color?

RESOLVED: The alpha component of the secondary color has generally been treated as zero. This extension specifies that only the R, G, and B components are added in the color sum operation, making the alpha component of the secondary color irrelevant. Other downstream extensions may allow applications to make use of this component.

(44) How are edge flags handled?

RESOLVED: Edge flags are passed through without the ability to be modified by a vertex program. Applications are free to send edge flags when vertex program mode is enabled.

(45) Should programs be C-style null-terminated strings?

RESOLVED: No. Programs should be specified as an array of GLubyte with an explicit length parameter. OpenGL has no precedent for passing null-terminated strings into the API (though GetString returns null-terminated strings). Null-terminated strings may be problematic for some programming languages.

(46) Should all existing OpenGL transform functionality and extensions be implementable as vertex programs?

RESOLVED: Yes. Vertex programs should be a complete superset of what you can do with OpenGL 1.2 and existing vertex transform extensions. To implement EXT_point_parameters, the VERTEX_PROGRAM_POINT_SIZE_ARB enable is introduced. To implement two-sided lighting, the VERTEX_PROGRAM_TWO_SIDE_ARB enable is introduced. To implement color material, applications should refer to the per-vertex color attribute in their vertex programs.

(47) Should there be a plural version of ProgramEnvParameter and ProgramLocalParameter, which would set multiple parameters in a single call?

RESOLVED: No; not necessary.

(48) Can the currently bound vertex program object be deleted or reloaded?

RESOLVED: Yes. When ProgramStringARB is called to reload a program object, subsequent program executions will use the new program. When DeleteProgramsARB deletes a currently bound program object, object zero becomes the new current program object.

(49) What happens if you transform vertices in vertex program mode, but the current program object does not contain a valid vertex program?

RESOLVED: Begin will fail with an INVALID_OPERATION error if the currently bound vertex program object does not have a valid program. The same applies to RasterPos and any command (Rect, DrawArrays, DrawElements) that implies a Begin.

Because Vertex is ignored outside of a Begin/End pair (without generating an error) it is impossible to provoke a vertex program if the current vertex program object is nonexistent or invalid. Other per-vertex parameters (for examples those set by Color, Normal, and VertexAttrib*ARB when the attribute number is not zero) are allowed since they are legal outside of a Begin/End.

(50) Discussing matrices is confusing because of row-major versus column-major issues. Can you give an example of how a matrix is bound?

RESOLVED: Assume program matrix zero were loaded with the following code:

```
// When loaded, the first row is "1, 2, 3, 4", because of column-major
// (OpenGL spec) vs. row-major (C) differences.
GLfloat matrix[16] = { 1, 5, 9,  13,
                       2, 6, 10, 14,
                       3, 7, 11, 15,
                       4, 8, 12, 16 };
glMatrixMode(GL_MATRIX0_ARB);
glLoadMatrixf(matrix);
```

Then in the program

```
!!ARBvp1.0
PARAM mat1[4] = { state.matrix.program[0] };
PARAM mat2[4] = { state.matrix.program[0].transpose };
```

mat1[0] would have (1,2,3,4), mat1[3] would have (13,14,15,16), mat2[0] would have (1,5,9,13), and mat2[3] would have (4,8,12,16).

(51) Should the new vertex program-related enables push/pop with ENABLE_BIT?

RESOLVED: Yes. Pushing and popping enable bits is easy.

(52) Should all the vertex attribute state push/pop with CURRENT_BIT?

RESOLVED: Yes.

(53) Should all the vertex attrib vertex array state push/pop with CLIENT_VERTEX_ARRAY_BIT?

RESOLVED: Yes.

(55) Should we generate an INVALID_VALUE operation if updating a vertex attribute greater than MAX_VERTEX_ATTRIBS_ARB?

RESOLVED: Yes. The other option would be to leave the behavior undefined, as with MultiTexCoord() functions. An implementation could mask or modulo the vertex attribute index with MAX_VERTEX_ATTRIB_ARB if it were a power of two. This error check will be a minor performance issue with VertexAttrib*ARB() and VertexAttribArrayARB() calls. There will be no per-vertex overhead when using vertex arrays or display lists.

(56) Should writes to program environment or local parameters during a vertex program be supported?

RESOLVED. No. Writes to program parameter registers from within a vertex program would require the execution of vertex programs to be serialized with respect to each other. This would create a severe implementation penalty for pipelined or parallel vertex program execution implementations.

(58) Should program objects be shared among rendering contexts in the same manner as display lists and texture objects?

RESOLVED: Yes.

(60) Should there be a MatrixMode or ActiveTexture-style selector for vertex attributes?

RESOLVED: No. While this would reduce the number of enumerants used by this extensions, it would create programming a hassle in lots of cases. Consider having to change the vertex attribute mode to enable a set of vertex arrays.

(61) How should queries of vertex attribute arrays work?

RESOLVED: Add new get commands. Using the existing calls would require adding 6 sets of 16+ enumerants for current state and vertex attribute array state. That's too many new enumerants. Instead, add GetVertexAttribARB and GetVertexAttribPointervARB. GetVertexAttribARB will be used to query vertex attribute array state and the current values of the generic vertex attributes. Get and GetPointerv will not return vertex attribute array state and pointers.

(63) What should be said about rendering invariances?

RESOLVED: See the Appendix A additions below.

The justification for the two rules cited is to support multi-pass rendering when using vertex programs. Different rendering passes will likely use different programs so there must be some means of guaranteeing that two different programs can generate particular identical vertex results between different passes.

In practice, this does limit the type of vertex program implementations that are possible.

For example, consider a limited hardware implementation of vertex programs that uses a different floating-point implementation than the CPU's floating-point implementation. If the limited hardware implementation can only run small vertex programs (say the hardware provides on 4 temporary registers instead of the required 12), the implementation is incorrect and non-conformant if programs that only require 4 temporary registers use the vertex program hardware, but programs that require more than 4 temporary registers are implemented by the CPU.

This is a very important practical requirement. Consider a multi-pass rendering algorithm where one pass uses a vertex program that uses only 4 temporary registers, but a different pass uses a vertex program that uses 5 temporary registers. If two programs have instruction sequences that given the same input state compute identical resulting vertex positions, the multi-pass algorithm should generate identically positioned primitives for each pass. But given the non-conformant vertex program implementation described above, this could not be guaranteed.

This does not mean that schemes for splitting vertex program implementations between dedicated hardware and CPUs are impossible. If the CPU and dedicated vertex program hardware used IDENTICAL floating-point implementations and therefore generated exactly identical results, the above described could work.

While these invariance rules are vital for vertex programs operating correctly for multi-pass algorithms, there is no requirement that conventional OpenGL vertex transform mode will be invariant with vertex program mode. A multi-pass algorithm should not assume that one pass using vertex program mode and another pass using conventional GL vertex transform mode will generate identically positioned primitives.

Consider that while the conventional OpenGL vertex program mode is repeatable with itself, the exact procedure used to transform vertices is not specified nor is the procedure's precision specified. The GL specification indicates that vertex coordinates are transformed by the modelview matrix and then transformed by the projection matrix. Some implementations may perform this sequence of transformations exactly, but other implementations may transform vertex coordinates by the composite of the modelview and projection matrices (one matrix transform instead of two matrix transforms in sequence). Given this implementation flexibility, there is no way for a vertex program author to exactly duplicate the precise computations used by the conventional OpenGL vertex transform mode.

The guidance to OpenGL application programs is clear. If you are going to implement multi-pass rendering algorithms that require certain invariances between the multiple passes, choose either vertex program mode or the conventional OpenGL vertex transform mode for your rendering passes, but do not mix the two modes.

(64) Should there be a way to guarantee position invariance with respect to conventional vertex transformation?

RESOLVED: Yes. The "OPTION ARB_position_invariant" program option addresses this issue. This program option will be available on all implementations of this extension.

John Carmack advocated the need for this.

(65) Why must RCP of 1.0 always be 1.0?

RESOLVED: This is important for 3D graphics so that non-projective textures and orthogonal projections work as expected. Basically when q or w is 1.0, things should work as expected. Stronger requirements such as "RCP of -1.0 must always be -1.0" are encouraged, but there is no compelling reason to state such requirements explicitly as is the case for "RCP of 1.0 must always be 1.0".

(66) What happens when the source scalar value for the ARL instruction is an extremely large positive or negative floating-point value? Is there a problem mapping the value to a constrained integer range?

RESOLVED: In this extension, address registers are only useful for relative addressing. The range of offsets that can be added to an address register is limited (-64 to +63) and the set of valid array indices is also limited to MAX_PROGRAM_PARAMETERS_ARB. So, the set of floating-point values that needs to be handled properly is well-constrained.

(67) How do you perform a 3-component normalize in three instructions?

RESOLVED: As follows.

```
DP3 result.w, vector, vector;     # result.w = nx^2+ny^2+nz^2
RSQ result.w, result.w;           # result.w = 1/sqrt(nx^2+ny^2+nz^2)
MUL result.xyz, result.w, vector;
```

(69) How do you compute the determinant of a 3x3 matrix in three instructions?

RESOLVED: As follows.

```
#
# Determinant of | vec0.x  vec0.y  vec0.z | into result.
#                | vec1.x  vec1.y  vec1.z |
#                | vec2.x  vec2.y  vec2.z |
#
MUL result, vec1.zxyw, vec2.yzxw;
MAD result, vec1.yzxw, vec2.zxyw, -result;
DP3 result, vec0, result;
```

(70) How do you transform a vertex position by a 4x4 matrix and then perform a homogeneous divide?

```
  RESOLVED:  As follows.

    ATTRIB  pos = vertex.position;
    TEMP    result, temp;
    PARAM   mat[4] = { state.matrix.modelview };

    DP4     result.w, pos, mat[3];
    DP4     result.x, pos, mat[0];
    DP4     result.y, pos, mat[1];
    DP4     result.z, pos, mat[2];
    RCP     temp.w, result.w;
    MUL     result, result, temp.w;
```

(71) How do you perform a vector weighting of two vectors using a single weight?

```
  RESOLVED:  As follows.

    # result = a * vec0 + (1-a) * vec1
    #        = vec1 + a * (vec0 - vec1)
    SUB result, vec0, vec1;
    MAD result, a, result, vec1;
```

*(72) How do you reduce a value to some fundamental period such as 2*PI?*

```
  RESOLVED:  As follows.

    # result = 2*PI * fraction(in/(2*PI))
    # piVec = (1/(2*PI), 2*PI, 0, 0)
    PARAM piVec = { 0.159154943, 6.283185307, 0, 0 };

    MUL result, in, piVec.x;
    EXP result, result.x;
    MUL result, result.y, piVec.y;
```

(73) How do you implement a simple ambient, specular, and diffuse infinite lighting computation with a single light and an eye-space normal?

```
  RESOLVED:  As follows.

    !!ARBvp1.0
    ATTRIB iPos         = vertex.position;
    ATTRIB iNormal      = vertex.normal;
    PARAM  mvinv[4]     = { state.matrix.modelview.invtrans };
    PARAM  mvp[4]       = { state.matrix.mvp };
    PARAM  lightDir     = state.light[0].position;
    PARAM  halfDir      = state.light[0].half;
    PARAM  specExp      = state.material.shininess;
    PARAM  ambientCol   = state.lightprod[0].ambient;
    PARAM  diffuseCol   = state.lightprod[0].diffuse;
    PARAM  specularCol  = state.lightprod[0].specular;
    TEMP   xfNormal, temp, dots;
    OUTPUT oPos         = result.position;
    OUTPUT oColor       = result.color;

    # Transform the vertex to clip coordinates.
    DP4   oPos.x, mvp[0], iPos;
    DP4   oPos.y, mvp[1], iPos;
    DP4   oPos.z, mvp[2], iPos;
    DP4   oPos.w, mvp[3], iPos;

    # Transform the normal to eye coordinates.
    DP3   xfNormal.x, mvinv[0], iNormal;
    DP3   xfNormal.y, mvinv[1], iNormal;
    DP3   xfNormal.z, mvinv[2], iNormal;

    # Compute diffuse and specular dot products and use LIT to compute
    # lighting coefficients.
    DP3   dots.x, xfNormal, lightDir;
    DP3   dots.y, xfNormal, halfDir;
    MOV   dots.w, specExp.x;
    LIT   dots, dots;

    # Accumulate color contributions.
    MAD   temp, dots.y, diffuseCol, ambientCol;
    MAD   oColor.xyz, dots.z, specularCol, temp;
    MOV   oColor.w, diffuseCol.w;
    END
```

(75) Can you perturb transformed vertex positions with a vertex program?

RESOLVED: Yes. Here is an example that performs an object-space diffuse lighting computations and perturbs the vertex position based on this lighting result. Do not take this example too seriously.

```
!!ARBvp1.0
#
# Program environment parameters:
# c[0].xyz = normalized light direction in object-space
#
# outputs diffuse illumination for color and perturbed position
#
ATTRIB iPos         = vertex.position;
ATTRIB iNormal      = vertex.normal;
PARAM  mvp[4]       = { state.matrix.mvp };
PARAM  lightDir     = program.env[0];
PARAM  diffuseCol   = { 1, 1, 0, 1 };
TEMP   temp;
OUTPUT oPos         = result.position;
OUTPUT oColor       = result.color;

DP3   temp, lightDir, iNormal;
MUL   oColor.xyz, temp, diffuseCol;
MAX   temp, temp, 0;            # clamp dot product to zero
MUL   temp, temp, iNormal;      # align in direction of normal
MUL   temp, temp, 0.125;        # scale displacement by 1/8
SUB   temp, temp, iPos;         # perturb
DP4   oPos.x, mvp[0], temp;     # xform using perturbed position
DP4   oPos.y, mvp[1], temp;
DP4   oPos.z, mvp[2], temp;
DP4   oPos.w, mvp[3], temp;
END
```

(76) Should this extension provide any method for updating program parameters in a program itself?

RESOLVED: No. NV_vertex_program provided a special mechanism to do this using a "vertex state program" manually executed by calling ExecuteProgramNV. This capability has not proven itself particularly useful to date.

(78) Should there be a different ProgramStringARB call for every distinct program target? Arguably, 1D, 2D, and 3D textures each have their own TexImage command for specifying their image data.

RESOLVED: No. All program objects can/should be loaded with ProgramStringARB. We expect the string to be a sufficient to express any kind of programmability.

Moreover, the 1D, 2D, and 3D TexImage commands describe the image being specified as opposed to the texture target being updated. With cube map textures, there are six face texture targets that use the TexImage2D command but not with the TEXTURE_2D target.

(79) This extension introduces a collection of new matrices for use by vertex programs (and possibly other programs as well). What should these matrices be called?

RESOLVED: Program matrices. These matrices are referred to as "tracking matrices" in NV_vertex_program, but the functionality is equivalent.

(80) With ARB_vertex_blend and EXT_vertex_weighting, there are multiple modelview matrices. This extension provides a single "MVP" matrix, defined to be the product of modelview matrix 0 and the projection matrices. Should this extension instead provide one MVP matrix per modelview matrix?

RESOLVED: No. Providing multiple MVP matrices allows applications to do N transformations into clip space and then one weighting operation, instead of N transformations into eye space, a weighting operation, and then a single transformation into clip space. This would potentially save instructions, but this optimization would be of no value if the program did any other operations that required eye coordinates.

Note also that the MVP transformations are likely general 4x4 matrix multiplies (4 DP4 instructions per transform). On the other hand, object and eye coordinates are often 3D coordinates with a constant W of 1.0. So each transformation to eye coordinates may require only 3 DP4 instructions, in which case the comparison may be 4N instructions (clip weighting) vs. 3N+4 (eye weighting).

(81) Should variable declarations be allowed to be anywhere within the program body, or should they instead be required to be done at the beginning of the program? Should the possibility of branching in a future standard affect this resolution?

RESOLVED: Declarations will be allowed anywhere in the program text; the only ordering requirement is that the declaration of a variable must precede its use in the program text. Requiring up-front variable declarations may require multiple passes for applications that build programs on the fly.

While declarations can appear anywhere in the program body, they are not executable statements. Any corresponding bindings (including constant initializations) are resolved before the program executes. The bindings will be resolved even if a program were to "branch around" a declaration.

(82) Should address register variables be treated as vectors? If so, should a variable number of components (up to four) be supported by this extension?

RESOLVED: In the future, four-component address vectors may be supported, and vector notation is used for forward compatibility. Using this notation makes address registers consistent with all the other vector data types in this extension. However, support for more than one usable component will be left for future extensions, but could be added via a program option or in a new language revision (e.g., !!ARBvp2.0).

(83) Should program local parameters be logically connected to the program string or the program object?

RESOLVED: Program local parameters are properties of a program object. Their values persist even after a new program is loaded into the object. This model does allow applications to recompile the program in a given object based on certain rendering settings without having to re-initialize any state stored in the object.

(84) Should this extension provide a method to specify "anonymous" program local parameters and query an index into the program parameter array.

RESOLVED: No. It would be nice to declare a variable in a program such as

```
PARAM foo = program.local;  # note no index in the array
```

after which an application could query the location of "foo" in the program local parameter array. However, given that local parameters persist even across program loads, it would be difficult to specify what program local parameter "foo" would be assigned to.

(85) EXT_vertex_weighting provides a single vertex blend weight. ARB_vertex_blend generalizes this concept to a weight vector. Both pieces of state are specified separately, and could be thought of as distinct. Should distinct bindings be provided in this extension?

RESOLVED: No. No current implementation supports both extensions, but the vendors involved in this standardization process agree that the state should not be considered distinct. If an implementation supported both extensions, the APIs would modify the same state.

(86) Should this extension provide functionality for variable aliasing? If so, how should it be specified and what types of variables can be aliasesed?

RESOLVED: Yes, for all variable types. The syntax is a simple text replacement:

```
ALIAS a = b;
```

This functionality allows applications to "share" variables, and thereby exceed implementation-dependent limits on the number of variable declarations. This may be particularly significant for temporaries, where the limit on the number of variables may be fairly low.

(87) How do you determine whether a given program option is supported by the GL implementation?

RESOLVED: Program options may be introduced in OpenGL extensions and may be added to future OpenGL specifications. An option will be supported if and only if (1) the corresponding OpenGL extension appears in the implementation-dependent EXTENSIONS string or (2) the option is documented in the OpenGL specification version corresponding to the implementation's VERSION string.

The ARB_position_invariant option is provided by this extension, and will always be available (provided this extension is supported).

(88) What's the deal with binding the alpha component of light colors, fog colors, and material colors (other than diffuse)? They don't do anything, right?

RESOLVED: The GL state for these different colors includes alpha components, which will be returned by queries. However, in the conventional OpenGL pipeline, most of these alpha components are effectively ignored. However, since they are present in the GL state, they will be exposed in bindings. What is done with these alpha values in program mode is completely up to the vertex program.

Vertex programs need to be careful to ensure that the alpha component is computed correctly when evaluating lighting equations. When accumulating light contributions, it may be necessary to use write masks to disable writes to the alpha component.

(89) The LOG instruction takes the logarithm of the absolute value of its operand while the LG2 instruction takes the logarithm of the operand itself. In LG2, the logarithm of negative numbers is undefined.

RESOLVED: The LOG instruction is present for (1) compatibility with NV_vertex_program and DirectX 8 languages and (2) because it may outperform LG2 on some platforms. For compatibility, it is defined to behave identically to existing languages.

(90) With vertex programs, fog coordinates and point sizes can be computed on a per-vertex basis. How are the fog coordinates and point sizes associated with vertices introduced by clipping computed?

RESOLVED: Fog coordinates and point sizes for clipped vertices are computed by interpolating the computed values at the original vertices in exactly the same manner as colors and texture coordinates are interpolated in section 2.13.8 of the OpenGL 1.3 specification.

(91) Vertex programs support only RGBA colors, but do not support color index inputs or results. What happens if an application uses vertex programs in color index mode.

RESOLVED: The results of vertex program execution are undefined if the GL is in color index mode.

(92) Should automatic normalization of evaluated normals (AUTO_NORMAL) be supported when the GL is in vertex program mode?

RESOLVED: Automatic normalization of normals will be disabled in vertex program mode. The current vertex program can easily normalize the normal if required. This can lead to greater efficiency if the vertex program transforms the normal to another coordinate system such as eye-space with a transform that preserves vector length. Then a single normalize after transform is more efficient than normalizing after evaluation and normalizing again after transform. Conceptually, the normalize mandated for AUTO_NORMAL in section 5.1 is just one of the many transformation operations subsumed by vertex programs.

(93) This extension allows applications to name their own variables. What keywords should be reserved?

RESOLVED: Instruction names and declaration keywords (e.g., PARAM) will be reserved. Additionally, since attribute, parameter, and result bindings are allowed in the program text, the binding prefix keywords "vertex", "state", "program", and "result" are reserved to simplify parsing. This prevents the need to distinguish between "vertex.position" ("vertex" as a binding) and "vertex.xyzw" ("vertex" as a variable).

(94) When counting the number of program parameter bindings, multiple constant vectors with the same components are counted only once. How is this determined?

RESOLVED: The implementation does a numerical comparison after the specified constants are converted to an internal floating-point representation. Due to floating-point representation limits, such conversions are not always precise. Constants specified with different text that are "equivalent" (e.g., "12" and "1.2E+01") are not guaranteed to resolve to the same value. Additionally, constants that are not "equivalent" but have only small relative differences (e.g., "200000000" and "200000001") may end up resolving to the same value. Constants specified with the same text should always be identical.

(95) What characters are allowed in identifier names?

RESOLVED: Letters ("A"-"Z", "a"-"z"), numbers ("0"-"9"), underscores ("_"), and dollar signs ("$").

(96) How should future programmability extensions interact with this one?

RESOLVED: Future programmability extensions are expected to fall in one of two classes: (1) extensions that bring programmability to new sections and (2) extensions the extend existing programmability models. The former class should introduce a new program target; the latter class would extend the functionality of an existing target.

Recommendations for extensions introducing new program targets include:

* Re-use and reference the functionality specified in this extension (or in a future OpenGL specification incorporating this extension) as much as possible, to maintain a consistent model.

* Provide a program header allowing for easy identification and versioning of programs for the new target.

Recommendations for extensions modifying existing program targets include:

* The option mechanism (section 2.14.4.5) should be used to provide minor modifications to the program language.

* The program header/version string (section 2.14.2) should be used to provide major modifications to the language, or potentially to provide a commonly used collection of options. Program header

```
        string changes should be multi-vendor extensions as much as
        possible.

      * For portability, programs should not be allowed to use extended
        language features without specifying the corresponding program
        options or program header.
```

New Procedures and Functions

```
    void VertexAttrib1sARB(uint index, short x);
    void VertexAttrib1fARB(uint index, float x);
    void VertexAttrib1dARB(uint index, double x);
    void VertexAttrib2sARB(uint index, short x, short y);
    void VertexAttrib2fARB(uint index, float x, float y);
    void VertexAttrib2dARB(uint index, double x, double y);
    void VertexAttrib3sARB(uint index, short x, short y, short z);
    void VertexAttrib3fARB(uint index, float x, float y, float z);
    void VertexAttrib3dARB(uint index, double x, double y, double z);
    void VertexAttrib4sARB(uint index, short x, short y, short z, short w);
    void VertexAttrib4fARB(uint index, float x, float y, float z, float w);
    void VertexAttrib4dARB(uint index, double x, double y, double z, double w);
    void VertexAttrib4NubARB(uint index, ubyte x, ubyte y, ubyte z, ubyte w);

    void VertexAttrib1svARB(uint index, const short *v);
    void VertexAttrib1fvARB(uint index, const float *v);
    void VertexAttrib1dvARB(uint index, const double *v);
    void VertexAttrib2svARB(uint index, const short *v);
    void VertexAttrib2fvARB(uint index, const float *v);
    void VertexAttrib2dvARB(uint index, const double *v);
    void VertexAttrib3svARB(uint index, const short *v);
    void VertexAttrib3fvARB(uint index, const float *v);
    void VertexAttrib3dvARB(uint index, const double *v);
    void VertexAttrib4bvARB(uint index, const byte *v);
    void VertexAttrib4svARB(uint index, const short *v);
    void VertexAttrib4ivARB(uint index, const int *v);
    void VertexAttrib4ubvARB(uint index, const ubyte *v);
    void VertexAttrib4usvARB(uint index, const ushort *v);
    void VertexAttrib4uivARB(uint index, const uint *v);
    void VertexAttrib4fvARB(uint index, const float *v);
    void VertexAttrib4dvARB(uint index, const double *v);
    void VertexAttrib4NbvARB(uint index, const byte *v);
    void VertexAttrib4NsvARB(uint index, const short *v);
    void VertexAttrib4NivARB(uint index, const int *v);
    void VertexAttrib4NubvARB(uint index, const ubyte *v);
    void VertexAttrib4NusvARB(uint index, const ushort *v);
    void VertexAttrib4NuivARB(uint index, const uint *v);

    void VertexAttribPointerARB(uint index, int size, enum type,
                                boolean normalized, sizei stride,
                                const void *pointer);

    void EnableVertexAttribArrayARB(uint index);
    void DisableVertexAttribArrayARB(uint index);

    void ProgramStringARB(enum target, enum format, sizei len,
                          const void *string);
    void BindProgramARB(enum target, uint program);

    void DeleteProgramsARB(sizei n, const uint *programs);

    void GenProgramsARB(sizei n, uint *programs);

    void ProgramEnvParameter4dARB(enum target, uint index,
                                  double x, double y, double z, double w);
    void ProgramEnvParameter4dvARB(enum target, uint index,
                                   const double *params);
    void ProgramEnvParameter4fARB(enum target, uint index,
                                  float x, float y, float z, float w);
    void ProgramEnvParameter4fvARB(enum target, uint index,
                                   const float *params);

    void ProgramLocalParameter4dARB(enum target, uint index,
                                    double x, double y, double z, double w);
    void ProgramLocalParameter4dvARB(enum target, uint index,
                                     const double *params);
    void ProgramLocalParameter4fARB(enum target, uint index,
                                    float x, float y, float z, float w);
    void ProgramLocalParameter4fvARB(enum target, uint index,
                                     const float *params);

    void GetProgramEnvParameterdvARB(enum target, uint index,
                                     double *params);
    void GetProgramEnvParameterfvARB(enum target, uint index,
                                     float *params);

    void GetProgramLocalParameterdvARB(enum target, uint index,
                                       double *params);
    void GetProgramLocalParameterfvARB(enum target, uint index,
                                       float *params);

    void GetProgramivARB(enum target, enum pname, int *params);

    void GetProgramStringARB(enum target, enum pname, void *string);

    void GetVertexAttribdvARB(uint index, enum pname, double *params);
    void GetVertexAttribfvARB(uint index, enum pname, float *params);
    void GetVertexAttribivARB(uint index, enum pname, int *params);

    void GetVertexAttribPointervARB(uint index, enum pname, void **pointer);

    boolean IsProgramARB(uint program);
```

New Tokens

Accepted by the <cap> parameter of Disable, Enable, and IsEnabled, by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev, and by the <target> parameter of ProgramStringARB, BindProgramARB, ProgramEnvParameter4[df][v]ARB, ProgramLocalParameter4[df][v]ARB, GetProgramEnvParameter[df]vARB, GetProgramLocalParameter[df]vARB, GetProgramivARB, and GetProgramStringARB.

```
        VERTEX_PROGRAM_ARB                              0x8620
```

Accepted by the <cap> parameter of Disable, Enable, and IsEnabled, and by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

VERTEX_PROGRAM_POINT_SIZE_ARB	0x8642
VERTEX_PROGRAM_TWO_SIDE_ARB	0x8643
COLOR_SUM_ARB	0x8458

Accepted by the <format> parameter of ProgramStringARB:

PROGRAM_FORMAT_ASCII_ARB	0x8875

Accepted by the <pname> parameter of GetVertexAttrib[dfi]vARB:

VERTEX_ATTRIB_ARRAY_ENABLED_ARB	0x8622
VERTEX_ATTRIB_ARRAY_SIZE_ARB	0x8623
VERTEX_ATTRIB_ARRAY_STRIDE_ARB	0x8624
VERTEX_ATTRIB_ARRAY_TYPE_ARB	0x8625
VERTEX_ATTRIB_ARRAY_NORMALIZED_ARB	0x886A
CURRENT_VERTEX_ATTRIB_ARB	0x8626

Accepted by the <pname> parameter of GetVertexAttribPointervARB:

VERTEX_ATTRIB_ARRAY_POINTER_ARB	0x8645

Accepted by the <pname> parameter of GetProgramivARB:

PROGRAM_LENGTH_ARB	0x8627
PROGRAM_FORMAT_ARB	0x8876
PROGRAM_BINDING_ARB	0x8677
PROGRAM_INSTRUCTIONS_ARB	0x88A0
MAX_PROGRAM_INSTRUCTIONS_ARB	0x88A1
PROGRAM_NATIVE_INSTRUCTIONS_ARB	0x88A2
MAX_PROGRAM_NATIVE_INSTRUCTIONS_ARB	0x88A3
PROGRAM_TEMPORARIES_ARB	0x88A4
MAX_PROGRAM_TEMPORARIES_ARB	0x88A5
PROGRAM_NATIVE_TEMPORARIES_ARB	0x88A6
MAX_PROGRAM_NATIVE_TEMPORARIES_ARB	0x88A7
PROGRAM_PARAMETERS_ARB	0x88A8
MAX_PROGRAM_PARAMETERS_ARB	0x88A9
PROGRAM_NATIVE_PARAMETERS_ARB	0x88AA
MAX_PROGRAM_NATIVE_PARAMETERS_ARB	0x88AB
PROGRAM_ATTRIBS_ARB	0x88AC
MAX_PROGRAM_ATTRIBS_ARB	0x88AD
PROGRAM_NATIVE_ATTRIBS_ARB	0x88AE
MAX_PROGRAM_NATIVE_ATTRIBS_ARB	0x88AF
PROGRAM_ADDRESS_REGISTERS_ARB	0x88B0
MAX_PROGRAM_ADDRESS_REGISTERS_ARB	0x88B1
PROGRAM_NATIVE_ADDRESS_REGISTERS_ARB	0x88B2
MAX_PROGRAM_NATIVE_ADDRESS_REGISTERS_ARB	0x88B3
MAX_PROGRAM_LOCAL_PARAMETERS_ARB	0x88B4
MAX_PROGRAM_ENV_PARAMETERS_ARB	0x88B5
PROGRAM_UNDER_NATIVE_LIMITS_ARB	0x88B6

Accepted by the <pname> parameter of GetProgramStringARB:

PROGRAM_STRING_ARB	0x8628

Accepted by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

PROGRAM_ERROR_POSITION_ARB	0x864B
CURRENT_MATRIX_ARB	0x8641
TRANSPOSE_CURRENT_MATRIX_ARB	0x88B7
CURRENT_MATRIX_STACK_DEPTH_ARB	0x8640
MAX_VERTEX_ATTRIBS_ARB	0x8869
MAX_PROGRAM_MATRICES_ARB	0x862F
MAX_PROGRAM_MATRIX_STACK_DEPTH_ARB	0x862E

Accepted by the <name> parameter of GetString:

PROGRAM_ERROR_STRING_ARB	0x8874

Accepted by the <mode> parameter of MatrixMode:

MATRIX0_ARB	0x88C0
MATRIX1_ARB	0x88C1
MATRIX2_ARB	0x88C2
MATRIX3_ARB	0x88C3
MATRIX4_ARB	0x88C4
MATRIX5_ARB	0x88C5
MATRIX6_ARB	0x88C6
MATRIX7_ARB	0x88C7
MATRIX8_ARB	0x88C8
MATRIX9_ARB	0x88C9
MATRIX10_ARB	0x88CA
MATRIX11_ARB	0x88CB
MATRIX12_ARB	0x88CC
MATRIX13_ARB	0x88CD
MATRIX14_ARB	0x88CE
MATRIX15_ARB	0x88CF
MATRIX16_ARB	0x88D0
MATRIX17_ARB	0x88D1
MATRIX18_ARB	0x88D2
MATRIX19_ARB	0x88D3
MATRIX20_ARB	0x88D4
MATRIX21_ARB	0x88D5
MATRIX22_ARB	0x88D6
MATRIX23_ARB	0x88D7
MATRIX24_ARB	0x88D8
MATRIX25_ARB	0x88D9
MATRIX26_ARB	0x88DA
MATRIX27_ARB	0x88DB
MATRIX28_ARB	0x88DC
MATRIX29_ARB	0x88DD
MATRIX30_ARB	0x88DE
MATRIX31_ARB	0x88DF

Additions to Chapter 2 of the OpenGL 1.3 Specification (OpenGL Operation)

Modify Section 2.6, Begin/End Paradigm (p. 12)

(modify last paragraph, p. 12) ... In addition, a current normal, a current color, multiple current texture coordinate sets, and multiple generic vertex attributes may be used in processing each vertex. Normals are used by the GL in lighting calculations; the current normal is a three-dimensional vector that may be set by sending three coordinates that specify it. Texture coordinates determine how a texture image is mapped onto a primitive. Multiple sets of texture coordinates may be used to specify how multiple texture images are mapped onto a primitive. Generic vertex attributes do not have any specific function but can be used in vertex program mode (section 2.14) to compute final values for any data associated with a vertex.

Modify Section 2.6.3, GL Commands within Begin/End (p. 19)

(modify first paragraph of section, p. 19) The only GL commands that are allowed within any Begin/End pairs are the commands for specifying vertex coordinates, vertex color, normal coordinates, texture coordinates, and generic vertex attributes (Vertex, Color, Index, Normal, TexCoord, VertexAttrib*ARB), ...

Modify Section 2.7, Vertex Specification (p. 19)

(remove the "Finally" from the next-to-last paragraph, p. 20) There are several ways to set the current color. The GL stores both a current single-valued color index, and a current four-valued RGBA color. One

(add new paragraph before last paragraph of section, p. 21) Vertex programs (section 2.14) can access an array of four-component generic current vertex attributes. The first entry of this array is numbered zero, and the number of entries in the array is given by the implementation-dependent constant MAX_VERTEX_ATTRIBS_ARB. The commands

```
  void VertexAttrib{1234}{sfd}ARB(uint index, T coords);
  void VertexAttrib{123}{sfd}vARB(uint index, T coords);
  void VertexAttrib4{bsifd ubusui}vARB(uint index, T coords);
```

specify the current vertex attribute numbered <index>, whose components are named <x>, <y>, <z>, and <w>. The VertexAttrib1ARB family of commands sets the <x> coordinate to the provided single argument while setting <y> and <z> to 0 and <w> to 1. Similarly, VertexAttrib2ARB commands set <x> and <y> to the specified values, <z> to 0 and <w> to 1; VertexAttrib3ARB commands set <x>, <y>, and <z>, with <w> set to 1, and VertexAttrib4ARB commands set all four coordinates. The error INVALID_VALUE is generated if <index> is greater than or equal to MAX_VERTEX_ATTRIBS_ARB.

The commands

```
  void VertexAttrib4NubARB(uint index, T coords);
  void VertexAttrib4N{bsi ubusui}vARB(uint index, T coords);
```

also specify vertex attributes with fixed-point coordinates that are scaled to the range [0,1] or [-1,1], according to Table 2.6.

Setting generic vertex attribute zero specifies a vertex; the four vertex coordinates are taken from the values of attribute zero. A Vertex2, Vertex3, or Vertex4 command is completely equivalent to the corresponding VertexAttrib command with an index of zero. Setting any other generic vertex attribute updates the current values of the attribute. There are no current values for vertex attribute zero.

Implementations may, but do not necessarily, use the same storage for the current values of generic and certain conventional vertex attributes. When any generic vertex attribute other than zero is specified, the current values for the corresponding conventional attribute in Table X.1 become undefined. Additionally, when a conventional vertex attribute is specified, the current values for the corresponding generic vertex attribute in Table X.1 become undefined. For example, setting the current normal will leave generic vertex attribute 2 undefined, and vice versa.

Generic Attribute	Conventional Attribute	Conventional Attribute Command
0	vertex position	Vertex
1	vertex weights 0-3	WeightARB, VertexWeightEXT
2	normal	Normal
3	primary color	Color
4	secondary color	SecondaryColorEXT
5	fog coordinate	FogCoordEXT
6	-	-
7	-	-
8	texture coordinate set 0	MultiTexCoord(TEXTURE0, ...)
9	texture coordinate set 1	MultiTexCoord(TEXTURE1, ...)
10	texture coordinate set 2	MultiTexCoord(TEXTURE2, ...)
11	texture coordinate set 3	MultiTexCoord(TEXTURE3, ...)
12	texture coordinate set 4	MultiTexCoord(TEXTURE4, ...)
13	texture coordinate set 5	MultiTexCoord(TEXTURE5, ...)
14	texture coordinate set 6	MultiTexCoord(TEXTURE6, ...)
15	texture coordinate set 7	MultiTexCoord(TEXTURE7, ...)
8+n	texture coordinate set n	MultiTexCoord(TEXTURE0+n, ...)

Table X.1, Generic and Conventional Vertex Attribute Mappings. For each row, the current value of the conventional attribute becomes undefined when the corresponding generic attribute is set, and vice versa. Attribute zero corresponds to the vertex position and has no current state.

Setting any conventional vertex attribute not listed in Table X.1 (including vertex weights 4 and above, if supported) will not cause any generic vertex attribute to become undefined, and such attributes will not become undefined when any generic vertex attribute is set.

(modify the last paragraph in the section, p.21) The state required to support vertex specification consists of four floating-point numbers per texture unit to store the current texture coordinates s, t, r, and q, three floating-point numbers to store the three coordinates of the current normal, four floating-point values to store the current RGBA color, one floating-point value to store the current color index, and MAX_VERTEX_ATTRIBS_ARB-1 four-component floating-point vectors for generic vertex attributes. There is no notion of a current vertex, so no state is devoted to vertex coordinates or vertex attribute zero. The initial texture coordinates are (S,T,R,Q) = (0,0,0,1) for each texture unit. The initial current normal has coordinates (0,0,1). The initial RGBA color is (R,G,B,A) = (1,1,1,1). The initial color index is 1. The initial values for all generic vertex attributes are undefined.

Modify Section 2.8, Vertex Arrays (p. 21)

(modify first paragraph of section, p.21) The vertex specification commands described in section 2.7 accept data in almost any format, but their use requires many command executions to specify even simple geometry. Vertex data may also be placed into arrays that are stored in the client's address space. Blocks of data in these arrays may then be used to specify multiple geometric primitives through the execution of a single GL command. The client may specify up to 5 plus the values of MAX_TEXTURE_UNITS and MAX_VERTEX_ATTRIBS_ARB arrays: one each to store vertex coordinates, edge flags, colors, color indices, normals, one or more texture coordinate sets, and one or more generic vertex attributes. The commands

```
  ...

  void VertexAttribPointerARB(uint index, int size, enum type,
                              boolean normalized, sizei stride,
                              const void *pointer);
```

describe the locations and organizations...

(add after the first paragraph, p.22) The <index> parameter in the VertexAttribPointer command identifies the generic vertex attribute array being described. The error INVALID_VALUE is generated if <index> is greater than or equal to MAX_VERTEX_ATTRIBS_ARB. The <normalized> parameter in the VertexAttribPointer command identifies whether fixed-point types should be normalized when converted to floating-point. If <normalized> is TRUE, fixed-point data are converted as specified in Table 2.6; otherwise, the fixed-point values are converted directly.

(add after first paragraph, p.23) An individual generic vertex attribute array is enabled or disabled by calling one of

```
  void EnableVertexAttribArrayARB(uint index);
  void DisableVertexAttribArrayARB(uint index);
```

where <index> identifies the generic vertex attribute array to enable or disable. The error INVALID_VALUE is generated if <index> is greater than or equal to MAX_VERTEX_ATTRIBS_ARB.

(modify Table 2.4, p.23)

Command	Sizes	Normalized?	Types
VertexPointer	2,3,4	no	short, int, float, double
NormalPointer	3	yes	byte, short, int, float, double
ColorPointer	3,4	yes	byte, ubyte, short, ushort, int, uint, float, double
IndexPointer	1	no	ubyte, short, int, float, double
TexCoordPointer	1,2,3,4	no	short, int, float, double
EdgeFlagPointer	1	no	boolean
VertexAttribPointerARB	1,2,3,4	flag	byte, ubyte, short, ushort, int, uint, float, double
WeightPointerARB	>=1	yes	byte, ubyte, short, ushort, int, uint, float, double
VertexWeightPointerEXT	1	n/a	float
SecondaryColor-PointerEXT	3	yes	byte, ubyte, short, ushort, int, uint, float, double
FogCoordPointerEXT	1	n/a	float, double
MatrixIndexPointerARB	>=1	no	ubyte, ushort, uint

Table 2.4: Vertex array sizes (values per vertex) and data types. The "normalized" column indicates whether fixed-point types are accepted directly or normalized to [0,1] (for unsigned types) or [-1,1] (for singed types). For generic vertex attributes, fixed-point data are normalized if and only if the <normalized> flag is set.

(modify last paragraph, p.23) The command

```
  void ArrayElement(int i);
```

transfers the ith element of every enabled array to the GL. The effect of ArrayElement(i) is the same as the effect of the command sequence

```
       if (ARB_vertex_blend vertex weight array enabled) {
         Weight[type]vARB(vertex weight array size,
                          vertex weight array element i);
       }
       if (EXT_vertex_weighting vertex weight array enabled) {
         VertexWeight[type]vARB(vertex weight array element i);
       }
       if (normal array enabled) {
         Normal3[type]v(normal array element i);
       }
       if (color array enabled) {
         Color[size][type]v(color array element i);
       }
       if (secondary color array enabled) {
         SecondaryColor3[type]vEXT(secondary color array element i);
       }
       if (fog coordinate array enabled) {
         FogCoord[type]vEXT(fog coordinate array element i);
       }
       if (matrix index array enabled) {
         MatrixIndex[type]vARB(matrix index array size,
                               matrix index array element i);
       }
       for (j = 0; j < textureUnits; j++) {
         if (texture coordinate set j array enabled) {
           MultiTexCoord[size][type]v(TEXTURE0 + j,
                                      texture coordinate set j
                                      array element i);
       }
       if (color index array enabled) {
         Index[type]v(color index array element i);
       }
       if (edge flag array enabled) {
         EdgeFlagv(edge flag array element i);
       }
       for (j = 1; j < genericAttributes; j++) {
         if (generic vertex attribute j array enabled) {
           if (generic vertex attribute j array normalization flag
               is set, and type is not FLOAT or DOUBLE) {
             VertexAttrib[size]N[type]vARB(j, generic vertex attribute j
                                              array element i);
           } else {
             VertexAttrib[size][type]vARB(j, generic vertex attribute j
                                             array element i);
           }
         }
       }
       if (generic attribute array 0 enabled) {
         if (generic vertex attribute j array normalization flag
             is set, and type is not FLOAT or DOUBLE) {
           VertexAttrib[size]N[type]vARB(0, generic vertex attribute 0
                                            array element i);
         } else {
           VertexAttrib[size][type]vARB(0, generic vertex attribute 0
                                           array element i);
         }
       } else if (vertex array enabled) {
         Vertex[size][type]vARB(vertex array element i);
       }
```

where <textureUnits> and <genericAttributes> give the number of texture units and generic vertex attributes supported by the implementation, respectively. "[size]" and "[type]" correspond to the size and type of the corresponding array. For generic vertex attributes, it is assumed that a complete set of vertex attribute commands exists, even though not all such functions are provided by the GL. Both generic attribute array zero and the vertex array can specify a vertex if enabled, but only one such array is used. As described in section 2.7, setting a generic vertex attributes listed in Table X.1 will leave the corresponding conventional vertex attribute undefined, and vice versa.

(modify last paragraph of section, p.28) If the number of supported texture units (the value of MAX TEXTURE UNITS) is m and the number of supported generic vertex attributes (MAX_VERTEX_ATTRIBS_ARB) is n, then the client state required to implement vertex arrays consists of 5+m+n boolean enables, 5+m+n memory pointers, 5+m+n integer stride values, 4+m+n symbolic constants representing array types, 2+m+n integers representing values per element, and n boolean normalization flags. In the initial state, the enable values are each disabled, the memory pointers are each null, the strides are each zero, the array types are each FLOAT, the integers representing values per element are each four, and the normalization flags are disabled.

Modify Section 2.10, Coordinate Transformations (p. 29)

(add new paragraphs) Vertex attributes are transformed before the vertex is used to generate primitives for rasterization, establish a raster position, or generate vertices for selection or feedback. The attributes of each vertex are transformed using one of two vertex transformation modes. The first mode, described in this and subsequent sections, is GL's conventional vertex transformation model. The second mode, known as vertex program mode and described in section 2.14, transforms vertex attributes as specified in an application-supplied vertex program.

Vertex program mode is enabled and disabled, respectively, by

```
  void Enable(enum target);
```

and

```
  void Disable(enum target);
```

with <target> equal to VERTEX_PROGRAM_ARB. When vertex program mode is enabled, vertices are transformed by the currently bound vertex program as discussed in section 2.14.

When vertex program mode is disabled, vertices, normals, and texture coordinates are transformed before their coordinates are used to produce an image in the framebuffer. We begin with a description of how vertex coordinates are transformed and how the transformation is controlled in this case. The discussion that continues through section 2.13 applies when vertex program mode is disabled.

Modify Section 2.10.2, Matrices (p. 31)

(modify 1st paragraph) The projection matrix and model-view matrix are set and modified with a variety of commands. The affected matrix is determined by the current matrix mode. The current matrix mode is set with

```
  void MatrixMode(enum mode);
```

which takes one of the pre-defined constants TEXTURE, MODELVIEW, COLOR, PROJECTION, or MATRIX<i>_ARB as the argument. In the case of MATRIX<i>_ARB, <i> is an integer between 0 and <n>-1 indicating one of <n> program matrices where <n> is the value of the implementation defined constant MAX_PROGRAM_MATRICES_ARB. Such program matrices are described in section 2.14.6. TEXTURE is described later in section 2.10.2, and COLOR is described in section 3.6.3. If the current matrix mode is MODELVIEW, then matrix operations apply to the model-view matrix; if PROJECTION, then they apply to the projection matrix.

(modify last paragraph of section) The state required to implement transformations consists of a <n>-value integer indicating the current matrix mode (where <n> is 4 + the number of supported texture and program matrices), a stack of at least two 4x4 matrices for each of COLOR, PROJECTION, and TEXTURE with associated stack pointers, <n> stacks (where <n> is at least 8) of at least one 4x4 matrix for each MATRIX<i>_ARB with associated stack pointers, and a stack of at least 32 4x4 matrices with an associated stack pointer for MODELVIEW. Initially, there is only one matrix on each stack, and all matrices are set to the identity. The initial matrix mode is MODELVIEW. The initial value of ACTIVE_TEXTURE is TEXTURE0.

Modify Section 2.11, Clipping (p. 39)

(add to end of next-to-last paragraph, p. 40) ... User clipping is not supported in vertex program mode if the current program is not position-invariant (section 2.14.4.5.1). In this case, client-defined clip planes are always treated as disabled.

Modify Section 2.12, Current Raster Position (p. 42)

(modify fourth paragraph, p.42) The coordinates are treated as if they were specified in a Vertex command. If vertex program mode is enabled, the currently bound vertex program is executed, using the x, y, z, and w coordinates as the object coordinates of the vertex. Otherwise, the x, y, z, and w coordinates are transformed by the current model-view and projection matrices. These coordinates, along with current values, are used to generate a color and texture coordinates just as is done for a vertex. The color and texture coordinates produced using either method replace the color and texture coordinates stored in the current raster position's associated data. When in vertex program mode, the "x" component of the fog coordinate result replaces the current raster distance; otherwise, the distance from the origin of the eye coordinate system to the vertex as transformed by only the current model-view matrix replaces the current raster distance. The latter distance can be approximated (see section 3.10).

Rename and Modify Section 2.13.8, Color and Vertex Data Clipping (p.56)

(modify second paragraph, p.57) Texture coordinates, as well as fog coordinates and point sizes computed on a per-vertex basis, must also be clipped when a primitive is clipped. The method is exactly analogous to that used for color clipping.

Add New Section 2.14 and subsections (p. 57).

Section 2.14, Vertex Programs

The conventional GL vertex transformation model described in sections 2.10 through 2.13 is a configurable but essentially hard-wired sequence of per-vertex computations based on a canonical set of per-vertex parameters and vertex transformation related state such as transformation matrices, lighting parameters, and texture coordinate generation parameters. The general success and utility of the conventional GL vertex transformation model reflects its basic correspondence to the typical vertex transformation requirements of 3D applications.

However when the conventional GL vertex transformation model is not sufficient, the vertex program mode provides a substantially more flexible model for vertex transformation. The vertex program mode permits applications to define their own vertex programs.

A vertex program is a character string that specifies a sequence of operations to perform. Vertex program instructions are typically 4-component vector operations that operate on per-vertex attributes and program parameters. Vertex programs execute on a per-vertex basis and operate on each vertex completely independently from any other vertices. Vertex programs execute a finite fixed sequence of instructions with no branching or looping. Vertex programs execute without data hazards so results computed in one instruction can be used immediately afterwards. The result of a vertex program is a set of vertex result registers that becomes the set of transformed vertex attributes used during clipping and primitive assembly.

Vertex programs are defined to operate only in RGBA mode. The results of vertex program execution are undefined if the GL is in color index mode.

Section 2.14.1, Program Objects

The GL provides one or more program targets, each identifying a portion of the GL that can be controlled through application-specified programs. The program target for vertex programs is VERTEX_PROGRAM_ARB. Each program target has an associated program object, called the current program object. Each program target also has a default program object, which is initially the current program object.

Each program object has an associated program string. The command

```
  ProgramStringARB(enum target, enum format, sizei len,
                   const void *string);
```

updates the program string for the current program object for <target>. <format> describes the format of the program string, which must currently be PROGRAM_FORMAT_ASCII_ARB. <string> is a pointer to the array of bytes representing the program string being loaded, which need not be null-terminated. The length of the array is given by <len>. If <string> is null-terminated, <len> should not include the terminator.

When a program string is loaded, it is interpreted according to syntactic and semantic rules corresponding to the program target specified by <target>. If a program violates the syntactic or semantic restrictions of the program target, ProgramStringARB generates the error

INVALID_OPERATION.

Additionally, ProgramString will update the program error position (PROGRAM_ERROR_POSITION_ARB) and error string (PROGRAM_ERROR_STRING_ARB). If a program fails to load, the value of the program error position is set to the ubyte offset into the specified program string indicating where the first program error was detected. If the program fails to load because of a semantic restriction that is not detected until the program is fully scanned, the error position is set to the value of <len>. If a program loads successfully, the error position is set to the value negative one. The implementation-dependent program error string contains one or more error or warning messages. If a program loads succesfully, the error string may either contain warning messages or be empty.

Each program object has an associated array of program local parameters. The number and type of program local parameters is target- and implementation-dependent. For vertex programs, program local parameters are four-component floating-point vectors. The number of vectors is given by the implementation-dependent constant MAX_PROGRAM_LOCAL_PARAMETERS_ARB, which must be at least 96. The commands

```
  void ProgramLocalParameter4fARB(enum target, uint index,
                                  float x, float y, float z, float w);
  void ProgramLocalParameter4fvARB(enum target, uint index,
                                   const float *params);
  void ProgramLocalParameter4dARB(enum target, uint index,
                                  double x, double y, double z, double w);
  void ProgramLocalParameter4dvARB(enum target, uint index,
                                   const double *params);
```

update the values of the program local parameter numbered <index> belonging to the program object currently bound to <target>. For ProgramLocalParameter4fARB and ProgramLocalParameter4dARB, the four components of the parameter are updated with the values of <x>, <y>, <z>, and <w>, respectively. For ProgramLocalParameter4fvARB and ProgramLocalParameter4dvARB, the four components of the parameter are updated with the array of four values pointed to by <params>. The error INVALID_VALUE is generated if <index> is greater than or equal to the number of program local parameters supported by <target>.

Additionally, each program target has an associated array of program environment parameters. Unlike program local parameters, program environment parameters are shared by all program objects of a given target. The number and type of program environment parameters is target- and implementation-dependent. For vertex programs, program environment parameters are four-component floating-point vectors. The number of vectors is given by the implementation-dependent constant MAX_PROGRAM_ENV_PARAMETERS_ARB, which must be at least 96. The commands

```
  void ProgramEnvParameter4fARB(enum target, uint index,
                                float x, float y, float z, float w);
  void ProgramEnvParameter4fvARB(enum target, uint index,
                                 const float *params);
  void ProgramEnvParameter4dARB(enum target, uint index,
                                double x, double y, double z, double w);
  void ProgramEnvParameter4dvARB(enum target, uint index,
                                 const double *params);
```

update the values of the program environment parameter numbered <index> for the given program target <target>. For ProgramEnvParameter4fARB and ProgramEnvParameter4dARB, the four components of the parameter are updated with the values of <x>, <y>, <z>, and <w>, respectively. For ProgramEnvParameter4fvARB and ProgramEnvParameter4dvARB, the four components of the parameter are updated with the array of four values pointed to by <params>. The error INVALID_VALUE is generated if <index> is greater than or equal to the number of program environment parameters supported by <target>.

Each program target has a default program object. Additionally, named program objects can be created and operated upon. The name space for program objects is the positive integers and is shared by programs of all targets. The name zero is reserved by the GL.

A named program object is created by binding an unused program object name to a valid program target. The binding is effected by calling

```
  BindProgramARB(enum target, uint program);
```

with <target> set to the desired program target and <program> set to the unused program name. The resulting program object has a program target given by <target> and is assigned target-specific default values (see section 2.14.7 for vertex programs). BindProgramARB may also be used to bind an existing program object to a program target. If <program> is zero, the default program object for <target> is bound. If <program> is the name of an existing program object whose associated program target is <target>, the named program object is bound. The error INVALID_OPERATION is generated if <program> names an existing program object whose associated program target is anything other than <target>.

Programs objects are deleted by calling

```
  void DeleteProgramsARB(sizei n, const uint *programs);
```

<programs> contains <n> names of programs to be deleted. After a program object is deleted, its name is again unused. If a program object that is bound to any target is deleted, it is as though BindProgramARB is first executed with same target and a <program> of zero. Unused names in <programs> are silently ignored, as is the value zero.

The command

```
  void GenProgramsARB(sizei n, uint *programs);
```

returns <n> currently unused program names in <programs>. These names are marked as used, for the purposes of GenProgramsARB only, but objects are created only when they are first bound using BindProgramARB.

Section 2.14.2, Vertex Program Grammar and Semantic Restrictions

Vertex program strings are specified as an array of ASCII characters containing the program text. When a vertex program is loaded by a call to ProgramStringARB, the program string is parsed into a set of tokens possibly separated by whitespace. Spaces, tabs, newlines, carriage returns, and comments are considered whitespace. Comments begin with the

    character "#" and are terminated by a newline, a carriage return, or the
    end of the program array.

    The Backus-Naur Form (BNF) grammar below specifies the syntactically valid
    sequences for vertex programs. The set of valid tokens can be inferred
    from the grammar. The token "" represents an empty string and is used to
    indicate optional rules. A program is invalid if it contains any
    undefined tokens or characters.

    A vertex program is required to begin with the header string "!!ARBvp1.0",
    without any preceding whitespace. This string identifies the subsequent
    program text as a vertex program (version 1.0) that should be parsed
    according to the following grammar and semantic rules. Program string
    parsing begins with the character immediately following the header string.

```
<program>              ::= <optionSequence> <statementSequence> "END"

<optionSequence>       ::= <optionSequence> <option>
                         | ""

<option>               ::= "OPTION" <identifier> ";"

<statementSequence>    ::= <statementSequence> <statement>
                         | ""

<statement>            ::= <instruction> ";"
                         | <namingStatement> ";"

<instruction>          ::= <ARL_instruction>
                         | <VECTORop_instruction>
                         | <SCALARop_instruction>
                         | <BINSCop_instruction>
                         | <BINop_instruction>
                         | <TRIop_instruction>
                         | <SWZ_instruction>

<ARL_instruction>      ::= "ARL" <maskedAddrReg> "," <scalarSrcReg>

<VECTORop_instruction> ::= <VECTORop> <maskedDstReg> "," <swizzleSrcReg>

<VECTORop>             ::= "ABS"
                         | "FLR"
                         | "FRC"
                         | "LIT"
                         | "MOV"

<SCALARop_instruction> ::= <SCALARop> <maskedDstReg> "," <scalarSrcReg>

<SCALARop>             ::= "EX2"
                         | "EXP"
                         | "LG2"
                         | "LOG"
                         | "RCP"
                         | "RSQ"

<BINSCop_instruction>  ::= <BINSCop> <maskedDstReg> "," <scalarSrcReg> ","
                           <scalarSrcReg>
```

```
    <BINSCop>                ::= "POW"

    <BINop_instruction>      ::= <BINop> <maskedDstReg> ","
                                 <swizzleSrcReg> "," <swizzleSrcReg>

    <BINop>                  ::= "ADD"
                               | "DP3"
                               | "DP4"
                               | "DPH"
                               | "DST"
                               | "MAX"
                               | "MIN"
                               | "MUL"
                               | "SGE"
                               | "SLT"
                               | "SUB"
                               | "XPD"

    <TRIop_instruction>      ::= <TRIop> <maskedDstReg> ","
                                 <swizzleSrcReg> "," <swizzleSrcReg> ","
                                 <swizzleSrcReg>

    <TRIop>                  ::= "MAD"

    <SWZ_instruction>        ::= "SWZ" <maskedDstReg> "," <srcReg> ","
                                 <extendedSwizzle>

    <scalarSrcReg>           ::= <optionalSign> <srcReg> <scalarSuffix>

    <swizzleSrcReg>          ::= <optionalSign> <srcReg> <swizzleSuffix>

    <maskedDstReg>           ::= <dstReg> <optionalMask>

    <maskedAddrReg>          ::= <addrReg> <addrWriteMask>

    <extendedSwizzle>        ::= <extSwizComp> "," <extSwizComp> ","
                                   <extSwizComp> "," <extSwizComp>

    <extSwizComp>            ::= <optionalSign> <extSwizSel>

    <extSwizSel>             ::= "0"
                               | "1"
                               | <component>

    <srcReg>                 ::= <vertexAttribReg>
                               | <temporaryReg>
                               | <progParamReg>

    <dstReg>                 ::= <temporaryReg>
                               | <vertexResultReg>

    <vertexAttribReg>        ::= <establishedName>
                               | <vtxAttribBinding>

    <temporaryReg>           ::= <establishedName>
    <progParamReg>          ::= <progParamSingle>
                              | <progParamArray> "[" <progParamArrayMem> "]"
                              | <paramSingleItemUse>

    <progParamSingle>       ::= <establishedName>

    <progParamArray>        ::= <establishedName>

    <progParamArrayMem>     ::= <progParamArrayAbs>
                              | <progParamArrayRel>

    <progParamArrayAbs>     ::= <integer>

    <progParamArrayRel>     ::= <addrReg> <addrComponent> <addrRegRelOffset>

    <addrRegRelOffset>      ::= ""
                              | "+" <addrRegPosOffset>
                              | "-" <addrRegNegOffset>

    <addrRegPosOffset>      ::= <integer> from 0 to 63

    <addrRegNegOffset>      ::= <integer> from 0 to 64

    <vertexResultReg>       ::= <establishedName>
                              | <resultBinding>

    <addrReg>               ::= <establishedName>

    <addrComponent>         ::= "." "x"

    <addrWriteMask>         ::= "." "x"

    <scalarSuffix>          ::= "." <component>

    <swizzleSuffix>         ::= ""
                              | "." <component>
                              | "." <component> <component>
                                    <component> <component>

    <component>             ::= "x"
                              | "y"
                              | "z"
                              | "w"
    <optionalMask>         ::= ""
                             | "." "x"
                             | "." "y"
                             | "." "xy"
                             | "." "z"
                             | "." "xz"
                             | "." "yz"
                             | "." "xyz"
                             | "." "w"
                             | "." "xw"
                             | "." "yw"
                             | "." "xyw"
                             | "." "zw"
                             | "." "xzw"
                             | "." "yzw"
                             | "." "xyzw"

    <namingStatement>      ::= <ATTRIB_statement>
                             | <PARAM_statement>
                             | <TEMP_statement>
                             | <ADDRESS_statement>
                             | <OUTPUT_statement>
                             | <ALIAS_statement>

    <ATTRIB_statement>     ::= "ATTRIB" <establishName> "="
                                 <vtxAttribBinding>

    <vtxAttribBinding>     ::= "vertex" "." <vtxAttribItem>

    <vtxAttribItem>        ::= "position"
                             | "weight" <vtxOptWeightNum>
                             | "normal"
                             | "color" <optColorType>
                             | "fogcoord"
                             | "texcoord" <optTexCoordNum>
                             | "matrixindex" "[" <vtxWeightNum> "]"
                             | "attrib" "[" <vtxAttribNum> "]"

    <vtxAttribNum>         ::= <integer> from 0 to MAX_VERTEX_ATTRIBS_ARB-1

    <vtxOptWeightNum>      ::= ""
                             | "[" <vtxWeightNum> "]"

    <vtxWeightNum>         ::= <integer> from 0 to MAX_VERTEX_UNITS_ARB-1,
                                 must be divisible by four

    <PARAM_statement>      ::= <PARAM_singleStmt>
                             | <PARAM_multipleStmt>

    <PARAM_singleStmt>     ::= "PARAM" <establishName> <paramSingleInit>

    <PARAM_multipleStmt>   ::= "PARAM" <establishName> "[" <optArraySize> "]"
                                   <paramMultipleInit>
    <optArraySize>         ::= ""
                             | <integer> from 1 to MAX_PROGRAM_PARAMETERS_ARB
                               (maximum number of allowed program
                                parameter bindings)

    <paramSingleInit>      ::= "=" <paramSingleItemDecl>

    <paramMultipleInit>    ::= "=" "{" <paramMultInitList> "}"

    <paramMultInitList>    ::= <paramMultipleItem>
                             | <paramMultipleItem> "," <paramMultiInitList>

    <paramSingleItemDecl>  ::= <stateSingleItem>
                             | <programSingleItem>
                             | <paramConstDecl>

    <paramSingleItemUse>   ::= <stateSingleItem>
                             | <programSingleItem>
                             | <paramConstUse>

    <paramMultipleItem>    ::= <stateMultipleItem>
                             | <programMultipleItem>
                             | <paramConstDecl>

    <stateMultipleItem>    ::= <stateSingleItem>
                             | "state" "." <stateMatrixRows>

    <stateSingleItem>      ::= "state" "." <stateMaterialItem>
                             | "state" "." <stateLightItem>
                             | "state" "." <stateLightModelItem>
                             | "state" "." <stateLightProdItem>
                             | "state" "." <stateTexGenItem>
                             | "state" "." <stateFogItem>
                             | "state" "." <stateClipPlaneItem>
                             | "state" "." <statePointItem>
                             | "state" "." <stateMatrixRow>

    <stateMaterialItem>    ::= "material" <optFaceType> "." <stateMatProperty>

    <stateMatProperty>     ::= "ambient"
                             | "diffuse"
                             | "specular"
                             | "emission"
                             | "shininess"

    <stateLightItem>       ::= "light" "[" <stateLightNumber> "]" "."
                               <stateLightProperty>

    <stateLightProperty>   ::= "ambient"
                             | "diffuse"
                             | "specular"
                             | "position"
                             | "attenuation"
                             | "spot" "." <stateSpotProperty>
                             | "half"

    <stateSpotProperty>    ::= "direction"
    <stateLightModelItem>  ::= "lightmodel" <stateLModProperty>

    <stateLModProperty>    ::= "." "ambient"
                             | <optFaceType> "." "scenecolor"

    <stateLightProdItem>   ::= "lightprod" "[" <stateLightNumber> "]"
                               <optFaceType> "." <stateLProdProperty>

    <stateLProdProperty>   ::= "ambient"
                             | "diffuse"
                             | "specular"

    <stateLightNumber>     ::= <integer> from 0 to MAX_LIGHTS-1

    <stateTexGenItem>      ::= "texgen" <optTexCoordNum> "."
                               <stateTexGenType> "." <stateTexGenCoord>

    <stateTexGenType>      ::= "eye"
                             | "object"

    <stateTexGenCoord>     ::= "s"
                             | "t"
                             | "r"
                             | "q"

    <stateFogItem>         ::= "fog" "." <stateFogProperty>

    <stateFogProperty>     ::= "color"
                             | "params"

    <stateClipPlaneItem>   ::= "clip" "[" <stateClipPlaneNum> "]" "." "plane"

    <stateClipPlaneNum>    ::= <integer> from 0 to MAX_CLIP_PLANES-1

    <statePointItem>       ::= "point" "." <statePointProperty>

    <statePointProperty>   ::= "size"
                             | "attenuation"

    <stateMatrixRow>       ::= <stateMatrixItem> "." "row" "["
                                <stateMatrixRowNum> "]"

    <stateMatrixRows>      ::= <stateMatrixItem> <optMatrixRows>

    <optMatrixRows>        ::= ""
                             | "." "row" "[" <stateMatrixRowNum> ".."
                                <stateMatrixRowNum> "]"

    <stateMatrixRow>       ::= <stateMatrixItem> "." "row" "["
                                <stateMatrixRowNum> "]"

    <stateOptMatModifier>  ::= ""
                             | "." <stateMatModifier>
    <stateMatModifier>     ::= "inverse"
                             | "transpose"
                             | "invtrans"

    <stateMatrixRowNum>    ::= <integer> from 0 to 3

    <stateMatrixName>      ::= "modelview" <stateOptModMatNum>
                             | "projection"
                             | "mvp"
                             | "texture" <optTexCoordNum>
                             | "palette" "[" <statePaletteMatNum> "]"
                             | "program" "[" <stateProgramMatNum> "]"

    <stateOptModMatNum>    ::= ""
                             | "[" <stateModMatNum> "]"

    <stateModMatNum>       ::= <integer> from 0 to MAX_VERTEX_UNITS_ARB-1

    <statePaletteMatNum>   ::= <integer> from 0 to MAX_PALETTE_MATRICES_ARB-1

    <stateProgramMatNum>   ::= <integer> from 0 to MAX_PROGRAM_MATRICES_ARB-1

    <programSingleItem>    ::= <progEnvParam>
                             | <progLocalParam>

    <programMultipleItem>  ::= <progEnvParams>
                             | <progLocalParams>

    <progEnvParams>        ::= "program" "." "env"
                                 "[" <progEnvParamNums> "]"

    <progEnvParamNums>     ::= <progEnvParamNum>
                             | <progEnvParamNum> ".." <progEnvParamNum>

    <progEnvParam>         ::= "program" "." "env"
                                 "[" <progEnvParamNum> "]"

    <progLocalParams>      ::= "program" "." "local"
                                 "[" <progLocalParamNums> "]"

    <progLocalParamNums>   ::= <progLocalParamNum>
                             | <progLocalParamNum> ".." <progLocalParamNum>

    <progLocalParam>       ::= "program" "." "local"
                                 "[" <progLocalParamNum> "]"

    <progEnvParamNum>      ::= <integer> from 0 to
                               MAX_PROGRAM_ENV_PARAMETERS_ARB - 1

    <progLocalParamNum>    ::= <integer> from 0 to
                               MAX_PROGRAM_LOCAL_PARAMETERS_ARB - 1

    <paramConstDecl>       ::= <paramConstScalarDecl>
                             | <paramConstVector>

    <paramConstUse>        ::= <paramConstScalarUse>
                             | <paramConstVector>
    <paramConstScalarDecl> ::= <signedFloatConstant>

    <paramConstScalarUse>  ::= <floatConstant>

    <paramConstVector>     ::= "{" <signedFloatConstant> "}"
                             | "{" <signedFloatConstant> ","
                                   <signedFloatConstant> "}"
                             | "{" <signedFloatConstant> ","
                                   <signedFloatConstant> ","
                                   <signedFloatConstant> "}"
                             | "{" <signedFloatConstant> ","
                                   <signedFloatConstant> ","
                                   <signedFloatConstant> ","
                                   <signedFloatConstant> "}"

    <signedFloatConstant>  ::= <optionalSign> <floatConstant>

    <floatConstant>        ::= see text

    <optionalSign>         ::= ""
                             | "-"
                             | "+"

    <TEMP_statement>       ::= "TEMP" <varNameList>

    <ADDRESS_statement>    ::= "ADDRESS" <varNameList>

    <varNameList>          ::= <establishName>
                             | <establishName> "," <varNameList>

    <OUTPUT_statement>     ::= "OUTPUT" <establishName> "="
                                 <resultBinding>

    <resultBinding>        ::= "result" "." "position"
                             | "result" "." <resultColBinding>
                             | "result" "." "fogcoord"
                             | "result" "." "pointsize"
                             | "result" "." "texcoord" <optTexCoordNum>

    <resultColBinding>     ::= "color" <optFaceType> <optColorType>

    <optFaceType>          ::= ""
                             | "." "front"
                             | "." "back"

    <optColorType>         ::= ""
                             | "." "primary"
                             | "." "secondary"

    <optTexCoordNum>       ::= ""
                             | "[" <texCoordNum> "]"

    <texCoordNum>          ::= <integer> from 0 to MAX_TEXTURE_UNITS-1

    <ALIAS_statement>      ::= "ALIAS" <establishName> "="
                                 <establishedName>
```

```
<establishName>        ::= <identifier>

<establishedName>      ::= <identifier>

<identifier>           ::= see text
```

The <integer> rule matches an integer constant. The integer consists of a sequence of one or more digits ("0" through "9").

The <floatConstant> rule matches a floating-point constant consisting of an integer part, a decimal point, a fraction part, an "e" or "E", and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of one or more digits ("0" through "9"). Either the integer part or the fraction parts (not both) may be missing; either the decimal point or the "e" (or "E") and the exponent (not both) may be missing.

The <identifier> rule matches a sequence of one or more letters ("A" through "Z", "a" through "z"), digits ("0" through "9), underscores ("_"), or dollar signs ("$"); the first character must not be a number. Upper and lower case letters are considered different (names are case-sensitive). The following strings are reserved keywords and may not be used as identifiers:

    ABS, ADD, ADDRESS, ALIAS, ARL, ATTRIB, DP3, DP4, DPH, DST, END, EX2,
    EXP, FLR, FRC, LG2, LIT, LOG, MAD, MAX, MIN, MOV, MUL, OPTION, OUTPUT,
    PARAM, POW, RCP, RSQ, SGE, SLT, SUB, SWZ, TEMP, XPD, program, result,
    state, and vertex.

The error INVALID_OPERATION is generated if a vertex program fails to load because it is not syntactically correct or for one of the semantic restrictions described in the following sections.

A successfully loaded vertex program is parsed into a sequence of instructions. Each instruction is identified by its tokenized name. The operation of these instructions when executed is defined in section 2.14.5. A successfully loaded program string replaces the program string previously loaded into the specified program object. If the OUT_OF_MEMORY error is generated by ProgramStringARB, no change is made to the previous contents of the current program object.

Section 2.14.3, Vertex Program Variables

Vertex programs may access a number of different variables during their execution. The following sections define the variables that can be declared and used by a vertex program.

Explicit variable declarations allow a vertex program to establish a variable name that can be used to refer to a specified resource in subsequent instructions. A vertex program will fail to load if it declares the same variable name more than once or if it refers to a variable name that has not been previously declared in the program string.

Implicit variable declarations allow a vertex program to use the name of certain available resources by name.

Section 2.14.3.1, Vertex Attributes

Vertex program attribute variables are a set of four-component floating-point vectors holding the attributes of the vertex being processed. Vertex attribute variables are read-only during vertex program execution.

Vertex attribute variables can be declared explicitly using the <ATTRIB_statement> grammar rule, or implicitly using the <vtxAttribBinding> grammar rule in an executable instruction.

Each vertex attribute variable is bound to a single item of vertex state according to the <vtxAttrBinding> grammar rule. The set of GL state that can be bound to a vertex attribute variable is given in Table X.2. Vertex attribute variables are initialized at each vertex program invocation with the current values of the bound state.

Vertex Attribute Binding	Components	Underlying State
vertex.position	(x,y,z,w)	object coordinates
vertex.weight	(w,w,w,w)	vertex weights 0-3
vertex.weight[n]	(w,w,w,w)	vertex weights n-n+3
vertex.normal	(x,y,z,1)	normal
vertex.color	(r,g,b,a)	primary color
vertex.color.primary	(r,g,b,a)	primary color
vertex.color.secondary	(r,g,b,a)	secondary color
vertex.fogcoord	(f,0,0,1)	fog coordinate
vertex.texcoord	(s,t,r,q)	texture coordinate, unit 0
vertex.texcoord[n]	(s,t,r,q)	texture coordinate, unit n
vertex.matrixindex	(i,i,i,i)	vertex matrix indices 0-3
vertex.matrixindex[n]	(i,i,i,i)	vertex matrix indices n-n+3
vertex.attrib[n]	(x,y,z,w)	generic vertex attribute n

Table X.2: Vertex Attribute Bindings. The "Components" column indicates the mapping of the state in the "Underlying State" column. Values of "0" or "1" in the "Components" column indicate the constants 0.0 and 1.0, respectively. Bindings containing "[n]" require an integer value of <n> to select an individual item.

If a vertex attribute binding matches "vertex.position", the "x", "y", "z" and "w" components of the vertex attribute variable are filled with the "x", "y", "z", and "w" components, respectively, of the vertex position.

If a vertex attribute binding matches "vertex.normal", the "x", "y", and "z" components of the vertex attribute variable are filled with the "x", "y", and "z" components, respectively, of the vertex normal. The "w" component is filled with 1.

If a vertex attribute binding matches "vertex.color" or "vertex.color.primary", the "x", "y", "z", and "w" components of the vertex attribute variable are filled with the "r", "g", "b", and "a" components, respectively, of the vertex color.

If a vertex attribute binding matches "vertex.color.secondary", the "x", "y", "z", and "w" components of the vertex attribute variable are filled with the "r", "g", "b", and "a" components, respectively, of the vertex secondary color.

If a vertex attribute binding matches "vertex.fogcoord", the "x" component of the vertex attribute variable is filled with the vertex fog coordinate. The "y", "z", and "w" coordinates are filled with 0, 0, and 1, respectively.

If a vertex attribute binding matches "vertex.texcoord" or "vertex.texcoord[n]", the "x", "y", "z", and "w" components of the vertex attribute variable are filled with the "s", "t", "r", and "q" components, respectively, of the vertex texture coordinates for texture unit <n>. If "[n]" is omitted, texture unit zero is used.

If a vertex attribute binding matches "vertex.weight" or "vertex.weight[n]", the "x", "y", "z", and "w" components of the vertex attribute variable are filled with vertex weights <n> through <n>+3, respectively. If "[n]" is omitted, weights zero through three are used. For the purposes of this binding, all weights supported by the implementation but not set by the application are set to zero, including the extra derived weight corresponding to the fixed-function WEIGHT_SUM_UNITY_ARB enable. For components whose corresponding weight is not supported by the implementation (i.e., numbered MAX_VERTEX_UNITS_ARB or larger), "y" and "z" components are set to 0.0 and "w" components are set to 1.0. A vertex program will fail to load if a vertex attribute binding specifies a weight number <n> that is greater than or equal to MAX_VERTEX_UNITS_ARB or is not divisible by four.

If a vertex attribute binding matches "vertex.matrixindex" or "vertex.matrixindex[n]", the "x", "y", "z", and "w" components of the vertex attribute variable are filled with matrix indices <n> through <n>+3 of the vertex, respectively. If "[n]" is omitted, matrix indices zero through three are used. For components whose corresponding matrix index is not supported by the implementation (i.e., numbered MAX_VERTEX_UNITS_ARB or larger), "y", and "z" components are set to 0.0 and "w" components are set to 1.0. A vertex program will fail to load if an attribute binding specifies a matrix index number <n> that is greater than or equal MAX_VERTEX_UNITS_ARB or is not divisible by four.

If a vertex attribute binding matches "vertex.attrib[n]", the "x", "y", "z" and "w" components of the vertex attribute variable are filled with the "x", "y", "z", and "w" components, respectively, of generic vertex attribute <n>. Note that "vertex.attrib[0]" and "vertex.position" are equivalent.

As described in section 2.7, setting a generic vertex attribute may leave a corresponding conventional vertex attribute undefined, and vice versa. To prevent inadvertent use of attribute pairs with undefined attributes, a vertex program will fail to load if it binds both a conventional vertex attribute and a generic vertex attribute listed in the same row of Table X.2.1.

Conventional Attribute Binding	Generic Attribute Binding
vertex.position	vertex.attrib[0]
vertex.weight	vertex.attrib[1]
vertex.weight[0]	vertex.attrib[1]
vertex.normal	vertex.attrib[2]
vertex.color	vertex.attrib[3]
vertex.color.primary	vertex.attrib[3]
vertex.color.secondary	vertex.attrib[4]
vertex.fogcoord	vertex.attrib[5]
vertex.texcoord	vertex.attrib[8]
vertex.texcoord[0]	vertex.attrib[8]
vertex.texcoord[1]	vertex.attrib[9]
vertex.texcoord[2]	vertex.attrib[10]
vertex.texcoord[3]	vertex.attrib[11]
vertex.texcoord[4]	vertex.attrib[12]
vertex.texcoord[5]	vertex.attrib[13]
vertex.texcoord[6]	vertex.attrib[14]
vertex.texcoord[7]	vertex.attrib[15]
vertex.texcoord[n]	vertex.attrib[8+n]

Table X.2.1: Invalid Vertex Attribute Binding Pairs. Vertex programs may not bind both attributes listed in any row. The <n> in the last row matches the number of any valid texture unit.

Section 2.14.3.2, Vertex Program Parameters

Vertex program parameter variables are a set of four-component floating-point vectors used as constants during vertex program execution. Vertex program parameters retain their values across vertex program invocations, although their values can change between invocations due to GL state changes.

Single program parameter variables and arrays of program parameter variables can be declared explicitly using the <PARAM_statement> grammar rule. Single program parameter variables can also be declared implicitly using the <paramSingleItemUse> grammar rule in an executable instruction.

Each single program parameter variable is bound to a constant vector or to a GL state vector according to the <paramSingleInit> grammar rule. Individual items of a program parameter array are bound to constant vectors or GL state vectors according to the <programMultipleInit> grammar rule. The set of GL state that can be bound to program parameter variables are given in Tables X.3.1 through X.3.8.

Constant Bindings

A program parameter variable can be bound to a scalar or vector constant using the <paramConstDecl> grammar rule (explicit declarations) or the <paramConstUse> grammar rule (implicit declarations).

If a program parameter binding matches the <paramConstScalarDecl> or <paramConstScalarUse> grammar rules, the corresponding program parameter variable is bound to the vector (X,X,X,X), where X is the value of the specified constant. Note that the <paramConstScalarUse> grammar rule, used only in implicit declarations, allows only non-negative constants. This disambiguates cases like "-2", which could conceivably be taken to mean either the vector "(2,2,2,2)" with all components negated or "(-2,-2,-2,-2)" without negation. Only the former interpretation is allowed by the grammar.

If a program parameter binding matches <paramConstVector>, the corresponding program parameter variable is bound to the vector (X,Y,Z,W), where X, Y, Z, and W are the values corresponding to the first, second, third, and fourth match of <signedFloatConstant>. If fewer than four constants are specified, Y, Z, and W assume the values 0.0, 0.0, and 1.0, if their respective constants are not specified.

Program parameter variables initialized to constant values can never be modified.

Program Environment/Local Parameter Bindings

Binding	Components	Underlying State
program.env[a]	(x,y,z,w)	program environment parameter a
program.local[a]	(x,y,z,w)	program local parameter a
program.env[a..b]	(x,y,z,w)	program environment parameters a through b
program.local[a..b]	(x,y,z,w)	program local parameters a through b

Table X.3.1: Program Environment/Local Parameter Bindings. <a> and <b> indicate parameter numbers, where <a> must be less than or equal to <b>.

If a program parameter binding matches "program.env[a]" or "program.local[a]", the four components of the program parameter variable are filled with the four components of program environment parameter <a> or program local parameter <a>, respectively.

Additionally, for program parameter array bindings, "program.env[a..b]" and "program.local[a..b]" are equivalent to specifying program environment parameters <a> through <b> in order or program local parameters <a> through <b> in order, respectively. In either case, a program will fail to load if <a> is greater than <b>.

Material Property Bindings

Binding	Components	Underlying State
state.material.ambient	(r,g,b,a)	front ambient material color
state.material.diffuse	(r,g,b,a)	front diffuse material color
state.material.specular	(r,g,b,a)	front specular material color
state.material.emission	(r,g,b,a)	front emissive material color
state.material.shininess	(s,0,0,1)	front material shininess
state.material.front.ambient	(r,g,b,a)	front ambient material color
state.material.front.diffuse	(r,g,b,a)	front diffuse material color
state.material.front.specular	(r,g,b,a)	front specular material color
state.material.front.emission	(r,g,b,a)	front emissive material color
state.material.front.shininess	(s,0,0,1)	front material shininess
state.material.back.ambient	(r,g,b,a)	back ambient material color
state.material.back.diffuse	(r,g,b,a)	back diffuse material color
state.material.back.specular	(r,g,b,a)	back specular material color
state.material.back.emission	(r,g,b,a)	back emissive material color
state.material.back.shininess	(s,0,0,1)	back material shininess

Table X.3.2: Material Property Bindings. If a material face is not specified in the binding, the front property is used.

If a program parameter binding matches any of the material properties listed in Table X.3.2, the program parameter variable is filled according to the table. For ambient, diffuse, specular, or emissive colors, the "x", "y", "z", and "w" components are filled with the "r", "g", "b", and "a" components, respectively, of the corresponding material color. For material shininess, the "x" component is filled with the material's specular exponent, and the "y", "z", and "w" components are filled with 0, 0, and 1, respectively. Bindings containing ".back" refer to the back material; all other bindings refer to the front material.

Material properties can be changed inside a Begin/End pair, either directly by calling Material, or indirectly through color material. However, such property changes are not guaranteed to update program parameter bindings until the following End command. Program parameter variables bound to material properties changed inside a Begin/End pair are undefined until the following End command.

Light Property Bindings

Binding	Components	Underlying State
state.light[n].ambient	(r,g,b,a)	light n ambient color
state.light[n].diffuse	(r,g,b,a)	light n diffuse color
state.light[n].specular	(r,g,b,a)	light n specular color
state.light[n].position	(x,y,z,w)	light n position
state.light[n].attenuation	(a,b,c,e)	light n attenuation constants and spot light exponent
state.light[n].spot.direction	(x,y,z,c)	light n spot direction and cutoff angle cosine
state.light[n].half	(x,y,z,1)	light n infinite half-angle
state.lightmodel.ambient	(r,g,b,a)	light model ambient color
state.lightmodel.scenecolor	(r,g,b,a)	light model front scene color
state.lightmodel . front.scenecolor	(r,g,b,a)	light model front scene color
state.lightmodel . back.scenecolor	(r,g,b,a)	light model back scene color
state.lightprod[n].ambient	(r,g,b,a)	light n / front material ambient color product
state.lightprod[n].diffuse	(r,g,b,a)	light n / front material diffuse color product
state.lightprod[n].specular	(r,g,b,a)	light n / front material specular color product
state.lightprod[n]. front.ambient	(r,g,b,a)	light n / front material ambient color product
state.lightprod[n]. front.diffuse	(r,g,b,a)	light n / front material diffuse color product
state.lightprod[n]. front.specular	(r,g,b,a)	light n / front material specular color product
state.lightprod[n]. back.ambient	(r,g,b,a)	light n / back material ambient color product
state.lightprod[n]. back.diffuse	(r,g,b,a)	light n / back material diffuse color product
state.lightprod[n]. back.specular	(r,g,b,a)	light n / back material specular color product

Table X.3.3: Light Property Bindings. <n> indicates a light number.

If a program parameter binding matches "state.light[n].ambient", "state.light[n].diffuse", or "state.light[n].specular", the "x", "y", "z", and "w" components of the program parameter variable are filled with the "r", "g", "b", and "a" components, respectively, of the corresponding light color.

If a program parameter binding matches "state.light[n].position", the "x", "y", "z", and "w" components of the program parameter variable are filled with the "x", "y", "z", and "w" components, respectively, of the light position.

If a program parameter binding matches "state.light[n].attenuation", the "x", "y", and "z" components of the program parameter variable are filled with the constant, linear, and quadratic attenuation parameters of the specified light, respectively (section 2.13.1). The "w" component of the program parameter variable is filled with the spot light exponent of the specified light.

If a program parameter binding matches "state.light[n].spot.direction", the "x", "y", and "z" components of the program parameter variable are filled with the "x", "y", and "z" components of the spot light direction of the specified light, respectively (section 2.13.1). The "w" component of the program parameter variable is filled with the cosine of the spot light cutoff angle of the specified light.

If a program parameter binding matches "state.light[n].half", the "x", "y", and "z" components of the program parameter variable are filled with the x, y, and z components, respectively, of the normalized infinite half-angle vector

```
  h_inf = || P + (0, 0, 1) ||.
```

The "w" component is filled with 1. In the computation of h_inf, P consists of the x, y, and z coordinates of the normalized vector from the eye position P_e to the eye-space light position P_pli (section 2.13.1). h_inf is defined to correspond to the normalized half-angle vector when using an infinite light (w coordinate of the position is zero) and an infinite viewer (v_bs is FALSE). For local lights or a local viewer, h_inf is well-defined but does not match the normalized half-angle vector, which will vary depending on the vertex position.

If a program parameter binding matches "state.lightmodel.ambient", the "x", "y", "z", and "w" components of the program parameter variable are filled with the "r", "g", "b", and "a" components of the light model ambient color, respectively.

If a program parameter binding matches "state.lightmodel.scenecolor" or "state.lightmodel.front.scenecolor", the "x", "y", and "z" components of the program parameter variable are filled with the "r", "g", and "b" components respectively of the "front scene color"

```
  c_scene = a_cs * a_cm + e_cm,
```

where a_cs is the light model ambient color, a_cm is the front ambient material color, and e_cm is the front emissive material color. The "w" component of the program parameter variable is filled with the alpha component of the front diffuse material color. If a program parameter binding matches "state.lightmodel.back.scenecolor", a similar back scene color, computed using back-facing material properties, is used. The front and back scene colors match the values that would be assigned to vertices using conventional lighting if all lights were disabled.

If a program parameter binding matches anything beginning with "state.lightprod[n]", the "x", "y", and "z" components of the program parameter variable are filled with the "r", "g", and "b" components, respectively, of the corresponding light product. The three light product components are the products of the corresponding color components of the specified material property and the light color of the specified light (see Table X.3.3). The "w" component of the program parameter variable is filled with the alpha component of the specified material property.

Light products depend on material properties, which can be changed inside a Begin/End pair. Such property changes are not guaranteed to take effect until the following End command. Program parameter variables bound to light products whose corresponding material property changes inside a Begin/End pair are undefined until the following End command.

Texture Coordinate Generation Property Bindings

Binding	Components	Underlying State
state.texgen[n].eye.s	(a,b,c,d)	TexGen eye linear plane coefficients, s coord, unit n
state.texgen[n].eye.t	(a,b,c,d)	TexGen eye linear plane coefficients, t coord, unit n
state.texgen[n].eye.r	(a,b,c,d)	TexGen eye linear plane coefficients, r coord, unit n
state.texgen[n].eye.q	(a,b,c,d)	TexGen eye linear plane coefficients, q coord, unit n
state.texgen[n].object.s	(a,b,c,d)	TexGen object linear plane coefficients, s coord, unit n
state.texgen[n].object.t	(a,b,c,d)	TexGen object linear plane coefficients, t coord, unit n
state.texgen[n].object.r	(a,b,c,d)	TexGen object linear plane coefficients, r coord, unit n
state.texgen[n].object.q	(a,b,c,d)	TexGen object linear plane coefficients, q coord, unit n

Table X.3.4: Texture Coordinate Generation Property Bindings. "[n]" is optional -- texture unit <n> is used if specified; texture unit 0 is used otherwise.

If a program parameter binding matches a set of TexGen plane coefficients, the "x", "y", "z", and "w" components of the program parameter variable are filled with the coefficients p1, p2, p3, and p4, respectively, for object linear coefficients, and the coefficents p1', p2', p3', and p4', respectively, for eye linear coefficients (section 2.10.4).

Fog Property Bindings

Binding	Components	Underlying State
state.fog.color	(r,g,b,a)	RGB fog color (section 3.10)
state.fog.params	(d,s,e,r)	fog density, linear start and end, and 1/(end-start) (section 3.10)

Table X.3.5: Fog Property Bindings

If a program parameter binding matches "state.fog.color", the "x", "y", "z", and "w" components of the program parameter variable are filled with the "r", "g", "b", and "a" components, respectively, of the fog color (section 3.10).

If a program parameter binding matches "state.fog.params", the "x", "y", and "z" components of the program parameter variable are filled with the fog density, linear fog start, and linear fog end parameters (section 3.10), respectively. The "w" component is filled with 1/(end-start), where end and start are the linear fog end and start parameters, respectively.

Clip Plane Property Bindings

Binding	Components	Underlying State
state.clip[n].plane	(a,b,c,d)	clip plane n coefficients

Table X.3.6: Clip Plane Property Bindings. <n> specifies the clip plane number, and is required.

If a program parameter binding matches "state.clip[n].plane", the "x", "y", "z", and "w" components of the program parameter variable are filled with the coefficients p1', p2', p3', and p4', respectively, of clip plane <n> (section 2.11).

Point Property Bindings

Binding	Components	Underlying State
state.point.size	(s,n,x,f)	point size, min and max size clamps, and fade threshold (section 3.3)
state.point.attenuation	(a,b,c,1)	point size attenuation consts

Table X.3.7: Point Property Bindings

If a program parameter binding matches "state.point.size", the "x", "y", "z", and "w" components of the program parameter variable are filled with the point size, minimum point size, maximum point size, and fade threshold, respectively (section 3.3).

If a program parameter binding matches "state.point.attenuation", the "x", "y", and "z" components of the program parameter variable are filled with the constant, linear, and quadratic point size attenuation parameters (a, b, and c), respectively (section 3.3). The "w" component is filled with 1.

Matrix Property Bindings

Binding	Underlying State
* state.matrix.modelview[n]	modelview matrix n
state.matrix.projection	projection matrix
state.matrix.mvp	modelview-projection matrix
* state.matrix.texture[n]	texture matrix n
state.matrix.palette[n]	modelview palette matrix n
state.matrix.program[n]	program matrix n

Table X.3.8: Base Matrix Property Bindings. The "[n]" syntax indicates a specific matrix number. For modelview and texture matrices, a matrix number is optional, and matrix zero will be used if the matrix number is omitted. These base bindings may further be modified by a inverse/transpose selector and a row selector.

If the beginning of a program parameter binding matches any of the matrix binding names listed in Table X.3.8, the binding corresponds to a 4x4 matrix. If the parameter binding is followed by ".inverse", ".transpose", or ".invtrans" (<stateMatModifier> grammar rule), the inverse, transpose, or transpose of the inverse, respectively, of the matrix specified in Table X.3.8 is selected. Otherwise, the matrix specified in Table X.3.8 is selected. If the specified matrix is poorly-conditioned (singular or nearly so), its inverse matrix is undefined. The binding name "state.matrix.mvp" refers to the product of modelview matrix zero and the projection matrix, defined as

```
   MVP = P * M0,
```

where P is the projection matrix and M0 is modelview matrix zero.

If the selected matrix is followed by ".row[<a>]" (matching the <stateMatrixRow> grammar rule), the "x", "y", "z", and "w" components of the program parameter variable are filled with the four entries of row <a> of the selected matrix. In the example,

```
  PARAM m0 = state.matrix.modelview[1].row[0];
  PARAM m1 = state.matrix.projection.transpose.row[3];
```

the variable "m0" is set to the first row (row 0) of modelview matrix 1 and "m1" is set to the last row (row 3) of the transpose of the projection matrix.

For program parameter array bindings, multiple rows of the selected matrix can be bound via the <stateMatrixRows> grammar rule. If the selected matrix binding is followed by ".row[<a>..<b>]", the result is equivalent to specifying matrix rows <a> through <b>, in order. A program will fail to load if <a> is greater than <b>. If no row selection is specified (<optMatrixRows> matches ""), matrix rows 0 through 3 are bound in order. In the example,

```
  PARAM m2[] = { state.matrix.program[0].row[1..2] };
  PARAM m3[] = { state.matrix.program[0].transpose };
```

the array "m2" has two entries, containing rows 1 and 2 of program matrix zero, and "m3" has four entries, containing all four rows of the transpose of program matrix zero.

Program Parameter Arrays

A program parameter array variable can be declared explicitly by matching the <PARAM_multipleStmt> grammar rule. Programs can optionally specify the number of individual program parameters in the array, using the <optArraySize> grammar rule. Program parameter arrays may not be declared implicity.

Individual parameter variables in a program parameter array are bound to GL state vectors or constant vectors as specified by the grammar rule <paramMultInitList>. Each individual parameter in the array is bound in turn as described above.

The total number of entries in the array is equal to the number of parameters bound in the initializer list. A vertex program that specifies an array size (<optArraySize> matches <integer>) that does not match the number of parameter bindings in the initialization list will fail to load.

Program parameter array variables may be accessed using absolute addressing by matching the <progParamArrayAbs> grammar rule, or relative addressing by matching the <progParamArrayRel> grammar rule.

Array accesses using absolute addressing are checked against the limits of the array. If any vertex program instruction accesses a program parameter array using absolute addressing with an out-of-range index (greater than or equal to the size of the array), the vertex program will fail to load.

Individual state vectors can have no more than one unique binding in any given program. The GL will automatically combine multiple bindings of the same state vector into a single unique binding, except for the case where a state vector is bound multiple times in program parameter arrays accessed using relative addressing. A vertex program will fail to load if any GL state vector is bound multiple times in a single array accessed using relative addressing or bound once in two or more arrays accessed using relative addressing.

Section 2.14.3.3, Vertex Program Temporaries

Vertex program temporary variables are a set of four-component floating-point vectors used to hold temporary results during vertex program execution. Temporaries do not persist between program invocations, and are undefined at the beginning of each vertex program invocation.

Vertex program temporary variables can be declared explicitly using the <TEMP_statement> grammar rule. Each such statement can declare one or more temporaries. Vertex program temporary variables can not be declared implicitly.

Section 2.14.3.4, Vertex Program Results

Vertex program result variables are a set of four-component floating-point vectors used to hold the final results of a vertex program. Vertex program result variables are write-only during vertex program execution.

Vertex program result variables can be declared explicitly using the <OUTPUT_statement> grammar rule, or implicitly using the <resultBinding> grammar rule in an executable instruction. Each vertex program result variable is bound to a transformed vertex attribute used during primitive assembly and rasterization. The set of vertex program result variable bindings is given in Table X.4.

Binding	Components	Description
result.position	(x,y,z,w)	position in clip coordinates
result.color	(r,g,b,a)	front-facing primary color
result.color.primary	(r,g,b,a)	front-facing primary color
result.color.secondary	(r,g,b,a)	front-facing secondary color
result.color.front	(r,g,b,a)	front-facing primary color
result.color.front.primary	(r,g,b,a)	front-facing primary color
result.color.front.secondary	(r,g,b,a)	front-facing secondary color
result.color.back	(r,g,b,a)	back-facing primary color
result.color.back.primary	(r,g,b,a)	back-facing primary color
result.color.back.secondary	(r,g,b,a)	back-facing secondary color
result.fogcoord	(f,*,*,*)	fog coordinate
result.pointsize	(s,*,*,*)	point size
result.texcoord	(s,t,r,q)	texture coordinate, unit 0
result.texcoord[n]	(s,t,r,q)	texture coordinate, unit n

Table X.4: Vertex Result Variable Bindings. Components labeled "*" are unused.

If a result variable binding matches "result.position", updates to the "x", "y", "z", and "w" components of the result variable modify the "x", "y", "z", and "w" components, respectively, of the transformed vertex's clip coordinates. Final window coordinates will be generated for the vertex as described in section 2.14.4.4.

If a result variable binding match begins with "result.color", updates to the "x", "y", "z", and "w" components of the result variable modify the "r", "g", "b", and "a" components, respectively, of the corresponding vertex color attribute in Table X.4. Color bindings that do not specify "front" or "back" are consided to refer to front-facing colors. Color bindings that do not specify "primary" or "secondary" are considered to refer to primary colors.

If a result variable binding matches "result.fogcoord", updates to the "x" component of the result variable set the transformed vertex's fog coordinate. Updates to the "y", "z", and "w" components of the result variable have no effect.

If a result variable binding matches "result.pointsize", updates to the "x" component of the result variable set the transformed vertex's point size. Updates to the "y", "z", and "w" components of the result variable have no effect.

If a result variable binding matches "result.texcoord" or "result.texcoord[n]", updates to the "x", "y", "z", and "w" components of the result variable set the "s", "t", "r" and "q" components, respectively, of the transformed vertex's texture coordinates for texture unit <n>. If "[n]" is omitted, texture unit zero is selected.

When in vertex program mode, all attributes of a transformed vertex are undefined at each vertex program invocation. Any results, or even individual components of results, that are not written to during vertex program execution remain undefined.

Section 2.14.3.5, Vertex Program Address Registers

Vertex program address register variables are a set of four-component signed integer vectors where only the "x" component of the address registers is currently accessible. Address registers are used as indices when performing relative addressing in program parameter arrays (section 2.14.4.2).

Vertex program address registers can be declared explicitly using the <ADDRESS_statement> grammar rule. Each such statement can declare one or more address registers. Vertex program address registers can not be declared implicitly.

Vertex program address register variables are undefined at each vertex program invocation. Address registers can be written by the ARL instruction (section 2.14.5.3), and will be read when a program uses relative addressing in program parameter arrays.

Section 2.14.3.6, Vertex Program Aliases

Vertex programs can create aliases by matching the <ALIAS_statement> grammar rule. Aliases allow programs to use multiple variable names to refer to a single underlying variable. For example, the statement

```
  ALIAS var1 = var0
```

establishes a variable name named "var1". Subsequent references to "var1" in the program text are treated as references to "var0". The left hand side of an ALIAS statement must be a new variable name, and the right hand side must be an established variable name.

Aliases are not considered variable declarations, so do not count against the limits on the number of variable declarations allowed in the program text.

Section 2.14.3.7, Vertex Program Resource Limits

The vertex program execution environment provides implementation-dependent resource limits on the number of instructions, temporary variable declarations, vertex attribute bindings, address register declarations, and program parameter bindings. A program that exceeds any of these resource limits will fail to load. The resource limits for vertex programs can be queried by calling GetProgramiv (section 6.1.12) with a target of VERTEX_PROGRAM_ARB.

The limit on vertex program instructions can be queried with a <pname> of MAX_PROGRAM_INSTRUCTIONS_ARB, and must be at least 128. Each instruction in the program (matching the <instruction> grammar rule) counts against this limit.

The limit on vertex program temporary variable declarations can be queried with a <pname> of MAX_PROGRAM_TEMPORARIES_ARB, and must be at least 12. Each temporary declared in the program, using the <TEMP_statement> grammar rule, counts against this limit. Aliases of declared temporaries do not.

The limit on vertex program attribute bindings can be queried with a <pname> of MAX_PROGRAM_ATTRIBS_ARB and must be at least 16. Each distinct vertex attribute bound explicitly or implicitly in the program counts against this limit; vertex attributes bound multiple times count only once.

The limit on vertex program address register declarations can be queried with a <pname> of MAX_PROGRAM_ADDRESS_REGISTERS_ARB, and must be at least 1. Each address register declared in the program, using the <ADDRESS_statement> grammar rule, counts against this limit.

The limit on vertex program parameter bindings can be queried with a <pname> of MAX_PROGRAM_PARAMETERS_ARB, and must be at least 96. Each distinct GL state vector bound explicitly or implicitly in the program counts against this limit; GL state vectors bound multiple times count only once. Each constant vector bound to an array accessed using relative addressing counts against this limit, even if the same constant vector is bound multiple times or in multiple arrays. Every other constant vector bound in the program is counted if and only if an identical constant vector has not already been counted. Two constant vectors are considered identical if the four component values are numerically equivalent. Recall that scalar constants bound in a program are treated as vector constants with the scalar value replicated. In the following code

```
    PARAM arr1[4] = { {1,2,3,4}, {1,2,3,4}, {4,4,4,4}, {5,6,7,8} };
    PARAM arr2[3] = { {1,2,3,4}, {5,6,7,8}, {0,1,2,3} };
    PARAM x = {4,3,2,1};
    PARAM y = {1,2,3,4};
    PARAM z = 4;
    PARAM r = {4,3,2,1};
```

assume that arr1 is accessed using relative addressing but arr2 is not. The four constants in arr1 all count against the limit. Only two other constants, {0,1,2,3} in arr2, and {4,3,2,1} in x, are counted; the other constants are identical to constants that had been previously counted.

In addition to the limits described above, the GL provides a similar set of implementation-dependent native resource limits. These limits, specified in section 6.1.12, provide guidance as to whether the program is small enough to use a "native" mode where vertex programs may be executed with higher performance. The native resource limits and usage counts are implementation-dependent and may not exactly correspond to limits and counts described above. In particular, native resource consumption may be reduced by program optimizations performed by the GL, or increased due to emulation of non-native instructions. Programs that satisfy the program resource limits described above, but whose native resource usage exceeds one or more native resource limits, are guaranteed to load but may execute suboptimally.

To assist in resource counting, the GL additionally provides GetProgram queries to determine the resource usage and native resource usage of the currently bound program, and to determine whether the bound program exceeds any native resource limit.

Section 2.14.4, Vertex Program Execution Environment

If vertex program mode is enabled, the currently bound vertex program is executed when a vertex is specified directly through the Vertex command, indirectly through vertex arrays or evaluators (section 5.1), or when the current raster position is updated.

If vertex program mode is enabled and the currently bound program object does not contain a valid vertex program, the error INVALID_OPERATION will be generated by Begin, RasterPos, and any command that implicitly calls Begin (e.g., DrawArrays).

Vertex programs execute a sequence of instructions without branching. Vertex programs begin by executing the first instruction in the program, and execute instructions in the order specified in the program until the last instruction is completed.

There are twenty-seven vertex program instructions. The instructions and their respective input and output parameters are summarized in Table X.5.

Instruction	Inputs	Output	Description
ABS	v	v	absolute value
ADD	v,v	v	add
ARL	s	a	address register load
DP3	v,v	ssss	3-component dot product
DP4	v,v	ssss	4-component dot product
DPH	v,v	ssss	homogeneous dot product
DST	v,v	v	distance vector
EX2	s	ssss	exponential base 2
EXP	s	v	exponential base 2 (approximate)
FLR	v	v	floor
FRC	v	v	fraction
LG2	s	ssss	logarithm base 2
LIT	v	v	compute light coefficients
LOG	s	v	logarithm base 2 (approximate)
MAD	v,v,v	v	multiply and add
MAX	v,v	v	maximum
MIN	v,v	v	minimum
MOV	v	v	move
MUL	v,v	v	multiply
POW	s,s	ssss	exponentiate
RCP	s	ssss	reciprocal
RSQ	s	ssss	reciprocal square root
SGE	v,v	v	set on greater than or equal
SLT	v,v	v	set on less than
SUB	v,v	v	subtract
SWZ	v	v	extended swizzle
XPD	v,v	v	cross product

Table X.5: Summary of vertex program instructions. "v" indicates a floating-point vector input or output, "s" indicates a floating-point scalar input, "ssss" indicates a scalar output replicated across a 4-component result vector, and "a" indicates a single address register component.

Section 2.14.4.1, Vertex Program Operands

Most vertex program instructions operate on floating-point vectors or scalars, as indicated by the grammar rules <swizzleSrcReg> and <scalarSrcReg>, respectively.

Vector and scalar operands can be obtained from vertex attribute, program parameter, or temporary registers, as indicated by the <srcReg> rule. For scalar operands, a single vector component is selected by the <scalarSuffix> rule, where the characters "x", "y", "z", and "w" select the x, y, z, and w components, respectively, of the vector.

Vector operands can be swizzled according to the <swizzleSuffix> rule. In its most general form, the <swizzleSuffix> rule matches the pattern ".????" where each question mark is replaced with one of "x", "y", "z", or "w". For such patterns, the x, y, z, and w components of the operand are taken from the vector components named by the first, second, third, and fourth character of the pattern, respectively. For example, if the swizzle suffix is ".yzzx" and the specified source contains {2,8,9,0}, the swizzled operand used by the instruction is {8,9,9,2}.

If the <swizzleSuffix> rule matches "", it is treated as though it were ".xyzw". If the <swizzleSuffix> rule matches (ignoring whitespace) ".x", ".y", ".z", or ".w", these are treated the same as ".xxxx", ".yyyy", ".zzzz", and ".wwww" respectively.

Floating-point scalar or vector operands can optionally be negated according to the <optionalSign> rule in <scalarSrcReg> and <swizzleSrcReg>. If the <optionalSign> matches "-", each operand or operand component is negated.

The following pseudo-code spells out the operand generation process. In the example, "float" is a floating-point scalar type, while "floatVec" is a four-component vector. "source" refers to the register used for the operand, matching the <srcReg> rule. "negate" is TRUE if the <optionalSign> rule in <scalarSrcReg> or <swizzleSrcReg> matches "-" and FALSE otherwise. The ".c***", ".*c**", ".**c*", ".***c" modifiers refer to the x, y, z, and w components obtained by the swizzle operation; the ".c" modifier refers to the single component selected for a scalar load.

```
  floatVec VectorLoad(floatVec source)
  {
      floatVec operand;

      operand.x = source.c***;
      operand.y = source.*c**;
      operand.z = source.**c*;
      operand.w = source.***c;
      if (negate) {
         operand.x = -operand.x;
         operand.y = -operand.y;
         operand.z = -operand.z;
         operand.w = -operand.w;
      }

      return operand;
  }
  float ScalarLoad(floatVec source)
  {
      float operand;

      operand = source.c;
      if (negate) {
        operand = -operand;
      }

      return operand;
  }
```

Section 2.14.4.2, Vertex Program Parameter Arrays

A vertex program can load a single element of a program parameter array using either absolute or relative addressing. Program parameter arrays are accessed when the <progParamArray> rule is matched.

Absolute addressing is used when the <progParamArrayMem> grammar rule matches <progParamArrayAbs>. When using absolute addressing, the offset of the selected entry in the array is given by the number matching <progParamRegNum>.

Relative addressing is used when the <progParamArrayMem> grammar rule matches <progParamArrayRel>. When using relative addressing, the offset of the selected entry in the array is computed by adding the address register component specified by the <addrReg> and <addrComponent> rules to the positive or negative offset specified by the <addrRegRelOffset> rule. If <addrRegRelOffset> matches "", no fixed offset is added to the address register component. If the computed offset is negative or exceeds the size of the array, the results of the access are undefined, but may not lead to program or GL termination.

The following pseudo-code spells out the process of loading a program parameter from an array. "addrReg" refers to the address register component used for relative addressing, "absolute" is TRUE if the operand uses absolute addressing and FALSE otherwise. "paramNumber" is the program parameter number for absolute addressing; "paramOffset" is the constant program parameter offset for relative addressing. "paramArray" is the parameter array that matches the <progParamArray> rule.

```
  floatVec ProgramParameterLoad(int addrReg)
  {
    int index;

    if (absolute) {
      index = paramNumber;
    } else {
      index = addrReg + paramOffset
    }

    return paramArray[index];
  }
```

Relative addressing can only be used for accessing program parameter arrays.

Section 2.14.4.3, Vertex Program Destination Register Update

Most vertex program instructions write a 4-component result vector to a single temporary or vertex result register. Writes to individual components of the destination register are controlled by individual component write masks specified as part of the instruction.

The component write mask is specified by the <optionalMask> rule found in the <maskedDstReg> rule. If the optional mask is "", all components are enabled. Otherwise, the optional mask names the individual components to enable. The characters "x", "y", "z", and "w" match the x, y, z, and w components respectively. For example, an optional mask of ".xzw" indicates that the x, z, and w components should be enabled for writing but the y component should not. The grammar requires that the destination register mask components must be listed in "xyzw" order.

Each component of the destination register is updated with the result of the vertex program instruction if and only if the component is enabled for writes by the component write mask. Otherwise, the component of the destination register remains unchanged.

The following pseudocode illustrates the process of writing a result vector to the destination register. In the pseudocode, "instrmask" refers to the component write mask given by the <optionalMask> rule. "result" and "destination" refer to the result vector and the register selected by <dstReg>, respectively.

```
  void UpdateDestination(floatVec destination, floatVec result)
  {
      floatVec merged;

      // Merge the converted result into the destination register, under
      // control of the compile-time write mask.
      merged = destination;
      if (instrMask.x) {
          merged.x = result.x;
      }
      if (instrMask.y) {
          merged.y = result.y;
      }
      if (instrMask.z) {
          merged.z = result.z;
      }
      if (instrMask.w) {
          merged.w = result.w;
      }

      // Write out the new destination register.
      destination = merged;
  }
```

The "ARL" instruction updates the single address register component similarly; the grammar is designed so that it writes to only the "x" component of an address register variable.

Section 2.14.4.4, Vertex Program Result Processing

As a vertex program executes, it will write to one or more result registers that are mapped to transformed vertex attributes. When a vertex program completes, the transformed vertex attributes are used to generate primitives.

The clip coordinates written to "result.position" are used to generate normalized device coordinates and window coordinates for the vertex in the manner described section 2.10.

Transformed vertices are then assembled into primitives and clipped as described in section 2.11.

The selection between front-facing and back-facing color attributes depends on the primitive to which the vertex belongs. If the primitive is a point or a line segment, or if vertex program two-sided color mode is disabled, the front-facing colors are always selected. If it is a polygon and two-sided color mode is enabled, then the selection is performed in exactly the same way as in two-sided lighting mode (section 2.13.1). Vertex program two-sided color mode is enabled and disabled by calling Enable or Disable with the symbolic value VERTEX_PROGRAM_TWO_SIDE_ARB.

Finally, as primitives are assembled, color clamping (section 2.13.6), flatshading (section 2.13.7), color, attribute clipping (section 2.13.8), and final color processing (section 2.13.9) operations are applied to the transformed vertices.

Section 2.14.4.5, Vertex Program Options

The <optionSequence> grammar rule provides a mechanism for programs to indicate that one or more extended language features are used by the program. All program options used by the program must be declared at the beginning of the program string. Each program option specified in a program string will modify the syntactic or semantic rules used to interpet the program and the execution environment used to execute the program. Program options not present in the program string are ignored, even if they are supported by the GL.

The <identifier> token in the <option> rule must match the name of a program option supported by the implementation. To avoid option name conflicts, option identifiers are required to begin with a vendor prefix. A program will fail to load if it specifies a program option not supported by the GL.

Vertex program options should confine their semantic changes to the domain of vertex programs. Support for a vertex program option should not change the specification and behavior of vertex programs not requesting use of that option.

2.14.4.5.1, Position-Invariant Vertex Program Option

If a vertex program specifies the "ARB_position_invariant" option, the program is used to generate all transformed vertex attributes except for position. Instead, clip coordinates are computed as specified in section 2.10. Additionally, user clipping is performed as described in section 2.11. Use of position-invariant vertex programs should generally guarantee that the transformed position of a vertex should be the same whether vertex program mode is enabled or disabled, allowing for correct mixed multi-pass rendering semantics.

When the position-invariant option is specified in a vertex program, vertex programs can no longer produced a transformed position. The <resultBinding> rule is modified to remove "result.position" from the list of token sequences matching the rule. A semantic restriction is added to indicate that a vertex program will fail to load if the number of instructions it contains exceeds the implementation-dependent limit minus four.

Section 2.14.5, Vertex Program Instruction Set

The following sections describe the set of supported vertex program instructions. Each section contains pseudocode describing the instruction. Instructions will have up to three operands, referred to as "op0", "op1", and "op2". The operands are loaded using the mechanisms specified in section 2.14.4.1. The variables "tmp", "tmp0", "tmp1", and "tmp2" describe scalars or vectors used to hold intermediate results in the instruction. Most instructions will generate a result vector called "result". The result vector is then written to the destination register specified in the instruction as described in section 2.14.4.3.

Section 2.14.5.1, ABS: Absolute Value

The ABS instruction performs a component-wise absolute value operation on the single operand to yield a result vector.

```
tmp = VectorLoad(op0);
result.x = fabs(tmp.x);
result.y = fabs(tmp.y);
result.z = fabs(tmp.z);
result.w = fabs(tmp.w);
```

Section 2.14.5.2, ADD: Add

The ADD instruction performs a component-wise add of the two operands to yield a result vector.

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1);
result.x = tmp0.x + tmp1.x;
result.y = tmp0.y + tmp1.y;
result.z = tmp0.z + tmp1.z;
result.w = tmp0.w + tmp1.w;
```

The following rules apply to addition:

1. <x> + <y> == <y> + <x>, for all <x> and <y>.
2. <x> + 0.0 == <x>, for all <x>.

Section 2.14.5.3, ARL: Address Register Load

The ARL instruction loads a single scalar operand and performs a floor operation to generate a signed integer scalar result:

```
    result = floor(ScalarLoad(op0));
```

The floor operation returns the largest integer less than or equal to the operand. For example floor(-1.7) = -2.0, floor(+1.0) = +1.0, and floor(+3.7) = +3.0.

Section 2.14.5.4, DP3: Three-Component Dot Product

The DP3 instruction computes a three-component dot product of the two operands (using the x, y, and z components) and replicates the dot product to all four components of the result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  dot = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
        (tmp0.z * tmp1.z);
  result.x = dot;
  result.y = dot;
  result.z = dot;
  result.w = dot;
```

Section 2.14.5.5, DP4: Four-Component Dot Product

The DP4 instruction computes a four-component dot product of the two operands and replicates the dot product to all four components of the result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1):
  dot = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
        (tmp0.z * tmp1.z) + (tmp0.w * tmp1.w);
  result.x = dot;
  result.y = dot;
  result.z = dot;
  result.w = dot;
```

Section 2.14.5.6, DPH: Homogeneous Dot Product

The DPH instruction computes a three-component dot product of the two operands (using the x, y, and z components), adds the w component of the second operand, and replicates the sum to all four components of the result vector. This is equivalent to a four-component dot product where the w component of the first operand is forced to 1.0.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1):
  dot = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
        (tmp0.z * tmp1.z) + tmp1.w;
  result.x = dot;
  result.y = dot;
  result.z = dot;
  result.w = dot;
```

Section 2.14.5.7, DST: Distance Vector

The DST instruction computes a distance vector from two specially-formatted operands. The first operand should be of the form [NA, d^2, d^2, NA] and the second operand should be of the form [NA, 1/d, NA, 1/d], where NA values are not relevant to the calculation and d is a vector length. If both vectors satisfy these conditions, the result vector will be of the form [1.0, d, d^2, 1/d].

The exact behavior is specified in the following pseudo-code:

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1);
result.x = 1.0;
result.y = tmp0.y * tmp1.y;
result.z = tmp0.z;
result.w = tmp1.w;
```

Given an arbitrary vector, d^2 can be obtained using the DP3 instruction (using the same vector for both operands) and 1/d can be obtained from d^2 using the RSQ instruction.

This distance vector is useful for per-vertex light attenuation calculations: a DP3 operation using the distance vector and an attenuation constants vector as operands will yield the attenuation factor.

Section 2.14.5.8, EX2: Exponential Base 2

The EX2 instruction approximates 2 raised to the power of the scalar operand and replicates the approximation to all four components of the result vector.

```
tmp = ScalarLoad(op0);
result.x = Approx2ToX(tmp);
result.y = Approx2ToX(tmp);
result.z = Approx2ToX(tmp);
result.w = Approx2ToX(tmp);
```

Section 2.14.5.9, EXP: Exponential Base 2 (approximate)

The EXP instruction computes a rough approximation of 2 raised to the power of the scalar operand. The approximation is returned in the "z" component of the result vector. A vertex program can also use the "x" and "y" components of the result vector to generate a more accurate approximation by evaluating

```
    result.x * f(result.y),
```

where f(x) is a user-defined function that approximates 2^x over the domain [0.0, 1.0). The "w" component of the result vector is always 1.0.

The exact behavior is specified in the following pseudo-code:

```
  tmp = ScalarLoad(op0);
  result.x = 2^floor(tmp);
  result.y = tmp - floor(tmp);
  result.z = RoughApprox2ToX(tmp);
  result.w = 1.0;
```

The approximation function is accurate to at least 10 bits:

```
  | RoughApprox2ToX(x) - 2^x | < 1.0 / 2^11, if 0.0 <= x < 1.0,
```

and, in general,

```
  | RoughApprox2ToX(x) - 2^x | < (1.0 / 2^11) * (2^floor(x)).
```

Section 2.14.5.10, FLR: Floor

The FLR instruction performs a component-wise floor operation on the operand to generate a result vector. The floor of a value is defined as the largest integer less than or equal to the value. The floor of 2.3 is 2.0; the floor of -3.6 is -4.0.

```
  tmp = VectorLoad(op0);
  result.x = floor(tmp.x);
  result.y = floor(tmp.y);
  result.z = floor(tmp.z);
  result.w = floor(tmp.w);
```

Section 2.14.5.11, FRC: Fraction

The FRC instruction extracts the fractional portion of each component of the operand to generate a result vector. The fractional portion of a component is defined as the result after subtracting off the floor of the component (see FLR), and is always in the range [0.0, 1.0).

For negative values, the fractional portion is NOT the number written to the right of the decimal point -- the fractional portion of -1.7 is not 0.7 -- it is 0.3. 0.3 is produced by subtracting the floor of -1.7 (-2.0) from -1.7.

```
  tmp = VectorLoad(op0);
  result.x = fraction(tmp.x);
  result.y = fraction(tmp.y);
  result.z = fraction(tmp.z);
  result.w = fraction(tmp.w);
```

Section 2.14.5.12, LG2: Logarithm Base 2

The LG2 instruction approximates the base 2 logarithm of the scalar operand and replicates it to all four components of the result vector.

```
  tmp = ScalarLoad(op0);
  result.x = ApproxLog2(tmp);
  result.y = ApproxLog2(tmp);
  result.z = ApproxLog2(tmp);
  result.w = ApproxLog2(tmp);
```

If the scalar operand is zero or negative, the result is undefined.

Section 2.14.5.13, LIT: Light Coefficients

The LIT instruction accelerates per-vertex lighting by computing lighting coefficients for ambient, diffuse, and specular light contributions. The "x" component of the single operand is assumed to hold a diffuse dot product (n dot VP_pli, as in the vertex lighting equations in Section 2.13.1). The "y" component of the operand is assumed to hold a specular dot product (n dot h_i). The "w" component of the operand is assumed to hold the specular exponent of the material (s_rm), and is clamped to the range (-128, +128) exclusive.

The "x" component of the result vector receives the value that should be multiplied by the ambient light/material product (always 1.0). The "y" component of the result vector receives the value that should be multiplied by the diffuse light/material product (n dot VP_pli). The "z" component of the result vector receives the value that should be multiplied by the specular light/material product (f_i * (n dot h_i) ^ s_rm). The "w" component of the result is the constant 1.0.

Negative diffuse and specular dot products are clamped to 0.0, as is done in the standard per-vertex lighting operations. In addition, if the diffuse dot product is zero or negative, the specular coefficient is forced to zero.

```
  tmp = VectorLoad(op0);
  if (tmp.x < 0) tmp.x = 0;
  if (tmp.y < 0) tmp.y = 0;
  if (tmp.w < -(128.0-epsilon)) tmp.w = -(128.0-epsilon);
  else if (tmp.w > 128-epsilon) tmp.w = 128-epsilon;
  result.x = 1.0;
  result.y = tmp.x;
  result.z = (tmp.x > 0) ? RoughApproxPower(tmp.y, tmp.w) : 0.0;
  result.w = 1.0;
```

The exponentiation approximation function may be defined in terms of the base 2 exponentiation and logarithm approximation operations in the EXP and LOG instructions, where

```
  RoughApproxPower(a,b) = RoughApproxExp2(b * RoughApproxLog2(a)).
```

In particular, the approximation may not be any more accurate than the underlying EXP and LOG operations.

Also, since 0^0 is defined to be 1, RoughApproxPower(0.0, 0.0) will produce 1.0.

Section 2.14.5.14, LOG: Logarithm Base 2 (approximate)

The LOG instruction computes a rough approximation of the base 2 logarithm of the absolute value of the scalar operand. The approximation is returned in the "z" component of the result vector. A vertex program can also use the "x" and "y" components of the result vector to generate a more accurate approximation by evaluating

```
result.x + f(result.y),
```

where f(x) is a user-defined function that approximates 2^x over the domain [1.0, 2.0). The "w" component of the result vector is always 1.0.

The exact behavior is specified in the following pseudo-code:

```
tmp = fabs(ScalarLoad(op0));
result.x = floor(log2(tmp));
result.y = tmp / 2^(floor(log2(tmp)));
result.z = RoughApproxLog2(tmp);
result.w = 1.0;
```

Here, "floor(log2(tmp))" refers to the floor of the exact logarithm, which can be easily computed for standard floating-point representations. The approximation function is accurate to at least 10 bits:

```
| RoughApproxLog2(x) - log_2(x) | < 1.0 / 2^11.
```

Section 2.14.5.15, MAD: Multiply and Add

The MAD instruction performs a component-wise multiply of the first two operands, and then does a component-wise add of the product to the third operand to yield a result vector.

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1);
tmp2 = VectorLoad(op2);
result.x = tmp0.x * tmp1.x + tmp2.x;
result.y = tmp0.y * tmp1.y + tmp2.y;
result.z = tmp0.z * tmp1.z + tmp2.z;
result.w = tmp0.w * tmp1.w + tmp2.w;
```

The multiplication and addition operations in this instruction are subject to the same rules as described for the MUL and ADD instructions.

Section 2.14.5.16, MAX: Maximum

The MAX instruction computes component-wise maximums of the values in the two operands to yield a result vector.

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1);
result.x = (tmp0.x > tmp1.x) ? tmp0.x : tmp1.x;
result.y = (tmp0.y > tmp1.y) ? tmp0.y : tmp1.y;
result.z = (tmp0.z > tmp1.z) ? tmp0.z : tmp1.z;
result.w = (tmp0.w > tmp1.w) ? tmp0.w : tmp1.w;
```

Section 2.14.5.17, MIN: Minimum

The MIN instruction computes component-wise minimums of the values in the two operands to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x > tmp1.x) ? tmp1.x : tmp0.x;
  result.y = (tmp0.y > tmp1.y) ? tmp1.y : tmp0.y;
  result.z = (tmp0.z > tmp1.z) ? tmp1.z : tmp0.z;
  result.w = (tmp0.w > tmp1.w) ? tmp1.w : tmp0.w;
```

Section 2.14.5.18, MOV: Move

The MOV instruction copies the value of the operand to yield a result vector.

```
  result = VectorLoad(op0);
```

Section 2.14.5.19, MUL: Multiply

The MUL instruction performs a component-wise multiply of the two operands to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = tmp0.x * tmp1.x;
  result.y = tmp0.y * tmp1.y;
  result.z = tmp0.z * tmp1.z;
  result.w = tmp0.w * tmp1.w;
```

The following rules apply to multiplication:

1. <x> * <y> == <y> * <x>, for all <x> and <y>.
2. +/-0.0 * <x> = +/-0.0, at least for all <x> that correspond to representable numbers (IEEE "not a number" and "infinity" encodings may be exceptions).
3. +1.0 * <x> = <x>, for all <x>.

Multiplication by zero and one should be invariant, as it may be used to evaluate conditional expressions without branching.

Section 2.14.5.20, POW: Exponentiate

The POW instruction approximates the value of the first scalar operand raised to the power of the second scalar operand and replicates it to all four components of the result vector.

```
  tmp0 = ScalarLoad(op0);
  tmp1 = ScalarLoad(op1);
  result.x = ApproxPower(tmp0, tmp1);
  result.y = ApproxPower(tmp0, tmp1);
  result.z = ApproxPower(tmp0, tmp1);
  result.w = ApproxPower(tmp0, tmp1);
```

The exponentiation approximation function may be implemented using the base 2 exponentiation and logarithm approximation operations in the EX2 and LG2 instructions. In particular,

```
  ApproxPower(a,b) = ApproxExp2(b * ApproxLog2(a)).
```

Note that a logarithm may be involved even for cases where the exponent is an integer. This means that it may not be possible to exponentiate correctly with a negative base. In constrast, it is possible in a "normal" mathematical formulation to raise negative numbers to integral powers (e.g., (-3)^2== 9, and (-0.5)^-2==4).

Section 2.14.5.21, RCP: Reciprocal

The RCP instruction approximates the reciprocal of the scalar operand and replicates it to all four components of the result vector.

```
  tmp = ScalarLoad(op0);
  result.x = ApproxReciprocal(tmp);
  result.y = ApproxReciprocal(tmp);
  result.z = ApproxReciprocal(tmp);
  result.w = ApproxReciprocal(tmp);
```

The following rule applies to reciprocation:

1. ApproxReciprocal(+1.0) = +1.0.

Section 2.14.5.22, RSQ: Reciprocal Square Root

The RSQ instruction approximates the reciprocal of the square root of the absolute value of the scalar operand and replicates it to all four components of the result vector.

```
  tmp = fabs(ScalarLoad(op0));
  result.x = ApproxRSQRT(tmp);
  result.y = ApproxRSQRT(tmp);
  result.z = ApproxRSQRT(tmp);
  result.w = ApproxRSQRT(tmp);
```

Section 2.14.5.23, SGE: Set On Greater or Equal Than

The SGE instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operands is greater than or equal that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x >= tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y >= tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z >= tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w >= tmp1.w) ? 1.0 : 0.0;
```

Section 2.14.5.24, SLT: Set On Less Than

The SLT instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operand is less than that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x < tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y < tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z < tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w < tmp1.w) ? 1.0 : 0.0;
```

Section 2.14.5.25, SUB: Subtract

The SUB instruction performs a component-wise subtraction of the second operand from the first to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = tmp0.x - tmp1.x;
  result.y = tmp0.y - tmp1.y;
  result.z = tmp0.z - tmp1.z;
  result.w = tmp0.w - tmp1.w;
```

Section 2.14.5.26, SWZ: Extended Swizzle

The SWZ instruction loads the single vector operand, and performs a swizzle operation more powerful than that provided for loading normal vector operands to yield an instruction vector.

After the operand is loaded, the "x", "y", "z", and "w" components of the result vector are selected by the first, second, third, and fourth matches of the <extSwizComp> pattern in the <extendedSwizzle> rule.

A result component can be selected from any of the four components of the operand or the constants 0.0 and 1.0. The result component can also be optionally negated. The following pseudocode describes the component selection method. "operand" refers to the vector operand, "select" is an enumerant where the values ZERO, ONE, X, Y, Z, and W correspond to the <extSwizSel> rule matching "0", "1", "x", "y", "z", and "w", respectively.

"negate" is TRUE if and only if the <optionalSign> rule in <extSwizComp> matches "-".

```
  float ExtSwizComponent(floatVec operand, enum select, boolean negate)
  {
      float result;
      switch (select) {
        case ZERO:  result = 0.0; break;
        case ONE:   result = 1.0; break;
        case X:     result = operand.x; break;
        case Y:     result = operand.y; break;
        case Z:     result = operand.z; break;
        case W:     result = operand.w; break;
      }
      if (negate) {
        result = -result;
      }
      return result;
  }
```

The entire extended swizzle operation is then defined using the following pseudocode:

```
  tmp = VectorLoad(op0);
  result.x = ExtSwizComponent(tmp, xSelect, xNegate);
  result.y = ExtSwizComponent(tmp, ySelect, yNegate);
  result.z = ExtSwizComponent(tmp, zSelect, zNegate);
  result.w = ExtSwizComponent(tmp, wSelect, wNegate);
```

"xSelect", "xNegate", "ySelect", "yNegate", "zSelect", "zNegate", "wSelect", and "wNegate" correspond to the "select" and "negate" values above for the four <extSwizComp> matches.

Since this instruction allows for component selection and negation for each individual component, the grammar does not allow the use of the normal swizzle and negation operations allowed for vector operands in other instructions.

Section 2.14.5.27, XPD: Cross Product

The XPD instruction computes the cross product using the first three components of its two vector operands to generate the x, y, and z components of the result vector. The w component of the result vector is undefined.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = tmp0.y * tmp1.z - tmp0.z * tmp1.y;
  result.y = tmp0.z * tmp1.x - tmp0.x * tmp1.z;
  result.z = tmp0.x * tmp1.y - tmp0.y * tmp1.x;
```

Section 2.14.6, Program Matrices

In addition to GL's conventional matrices, several additional program matrices are available for use as program parameters. These matrices have names of the form MATRIX<i>_ARB where <i> is between zero and <n>-1 where <n> is the value of the implementation-dependent constant

MAX_PROGRAM_MATRICES_ARB. The MATRIX<i>_ARB constants obey MATRIX<i>_ARB = MATRIX0_ARB + <i>. The value of MAX_PROGRAM_MATRICES_ARB must be at least eight. The maximum stack depth for program matrices is defined by the MAX_PROGRAM_MATRIX_STACK_DEPTH_ARB and must be at least 1.

Section 2.14.7 Required Vertex Program State

The state required to support program objects of all targets consists of:

an integer for the program error position, initially -1;

an array of ubytes for the program error string, initially empty;

and the state that must be maintained to indicate which integers are currently in use as program object names.

The state required to support the vertex program target consists of:

a bit indicating whether or not program mode is enabled, initially disabled;

a bit indicating whether or not vertex program two-sided color mode is enabled, initially disabled;

a bit indicating whether or not vertex program point size mode is enabled, initially disabled;

a set of MAX_PROGRAM_ENV_PARAMETERS_ARB four-component floating-point program environment parameters, initially set to (0,0,0,0);

and an unsigned integer naming the currently bound vertex program, initially zero.

The state required for each vertex program object consists of:

an unsigned integer indicating the program object name;

an array of type ubyte containing the program string, initially empty;

an unsigned integer holding the length of the program string, initially zero;

an enum indicating the program string format, initially PROGRAM_FORMAT_ASCII_ARB;

five unsigned integers holding the number of instruction, temporary variable, vertex attribute binding, address register, and program parameter binding resources used by the program, initially all zero;

five unsigned integers holding the number of native instruction, temporary variable, vertex attribute binding, address register, and program parameter binding resources used by the program, initially all zero;

and a set of MAX_PROGRAM_LOCAL_PARAMETERS_ARB four-component floating-point program local parameters, initially set to (0,0,0,0).

Initially, no vertex program objects exist.

Additions to Chapter 3 of the OpenGL 1.3 Specification (Rasterization)

Modify Section 3.3, Points (p. 63)

(replace the first paragraph) When vertex program mode and vertex progam point size mode are both enabled, the point size used for point rasterization is taken from the transformed vertex's point size attribute. Otherwise, it is controlled with

```
void PointSize(float size);
```

size specifies the width or diameter of a point. The initial point size value is 1.0. A value less than or equal to zero results in the error INVALID_VALUE.

Vertex program point size mode is enabled and disabled by calling Enable or Disable with the symbolic value VERTEX_PROGRAM_POINT_SIZE_ARB.

Modify Section 3.9, Color Sum (p. 154)

After texturing, a fragment has two RGBA colors: a primary color c_pri (which texturing, if enabled, may have modified) and a secondary color c_sec. If color sum is enabled, the R, G, and B components of these two colors are summed, and with the A component of the primary color produce a single post-texturing RGBA color c. The components of c are then clamped to the range [0,1]. If color sum is disabled, then c_pri is assigned to the post-texturing color.

Color sum is enabled or disabled using the generic Enable and Disable commands, respectively, with the symbolic constant COLOR_SUM_ARB. If vertex program mode is disabled and lighting is enabled, the color sum stage is always applied, ignoring the value of COLOR_SUM_ARB.

The state required is a single bit indicating whether color sum is enabled or disabled. In the initial state, color sum is disabled.

Modify Section 3.10, Fog (p. 154)

(modify second paragraph) This factor f may be computed according to one of three equations:

```
        f = exp(-d*c),                                (3.24)
        f = exp(-(d*c)^2), or                         (3.25)
        f = (e-c)/(e-s)                               (3.26)
```

If vertex program mode is enabled or if the fog source (as defined below) is FOG_COORDINATE_EXT, then c is the fragment's fog coordinate. Otherwise, the c is the eye-coordinate distance from the eye, (0,0,0,1) in eye-coordinates, to the fragment center. ...

Additions to Chapter 4 of the OpenGL 1.3 Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the OpenGL 1.3 Specification (Special Functions)

Modify Section 5.1, Evaluators (p. 181)

(modify next-to-last paragraph, p. 184) For MAP VERTEX 3, let q = p. For MAP VERTEX 4, let q=(x/w,y/w,z/w), where (x; y; z;w) = p. Then let

```
        dq   dq
    m = -- x --.
        du   dv
```

The the generated analytic normal, n, is given by n=m if vertex program mode is enabled or by n=m/|m| if vertex program mode is disabled.

Modify Section 5.4, Display Lists (p. 191)

(modify third paragraph, p. 195) ... These are IsList, GenLists, ..., IsProgramARB, GenProgramsARB, DeleteProgramsARB, and VertexAttribPointerARB, EnableVertexAttribArrayARB, DisableVertexAttribArrayARB, as well as IsEnabled and all the Get commands (chapter 6).

Additions to Chapter 6 of the OpenGL 1.3 Specification (State and State Requests)

Modify Section 6.1.2, Data Conversions (p. 198)

(add before last paragraph, p. 198) The matrix selected by the current matrix mode can be queried by calling GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev with <pname> set to CURRENT_MATRIX_ARB; the matrix will be returned in transposed form with <pname> set to TRANSPOSE_CURRENT_MATRIX_ARB. The depth of the selected matrix stack can be queried with <pname> set to CURRENT_MATRIX_STACK_DEPTH_ARB. Querying CURRENT_MATRIX_ARB and CURRENT_MATRIX_STACK_DEPTH_ARB is the only means for querying the matrix and matrix stack depth of the program matrices described in section 2.14.6.

Modify Section 6.1.11, Pointer and String Queries (p. 206)

(modify last paragraph, p. 206) ... The possible values for <name> are VENDOR, RENDERER, VERSION, EXTENSIONS, and PROGRAM_ERROR_STRING_ARB.

(add after last paragraph of section, p. 207) Queries of PROGRAM_ERROR_STRING_ARB return a pointer to an implementation-dependent program load error string. If the last call to ProgramStringARB failed to load a program, the returned string describes at least one reason why the program failed to load. If the last call to ProgramStringARB successfully loaded a program, the returned string may be empty (containing only a zero terminator) or may contain one or more implementation-dependent warning messages. The contents of the error string are guaranteed to remain constant only until the next ProgramStringARB command, which may overwrite the error string.

Insert a new Section 6.1.12, Program Queries (p. 207), between existing sections 6.1.11 and 6.1.12.

Section 6.1.12, Program Queries

The commands

```
  void GetProgramEnvParameterdvARB(enum target, uint index,
                                   double *params);
  void GetProgramEnvParameterfvARB(enum target, uint index,
                                   float *params);
```

obtain the current value for the program environment parameter numbered <index> for the given program target <target>, and places the information in the array <params>. The error INVALID_ENUM is generated if <target> specifies a nonexistent program target or a program target that does not support program environment parameters. The error INVALID_VALUE is generated if <index> is greater than or equal to the implementation-dependent number of supported program environment parameters for the program target.

When <target> is VERTEX_PROGRAM_ARB, each program parameter returned is an array of four values.

The commands

```
  void GetProgramLocalParameterdvARB(enum target, uint index,
                                     double *params);
  void GetProgramLocalParameterfvARB(enum target, uint index,
                                     float *params);
```

obtain the current value for the program local parameter numbered <index> belonging to the program object currently bound to <target>, and places the information in the array <params>. The error INVALID_ENUM is generated if <target> specifies a nonexistent program target or a program target that does not support program local parameters. The error INVALID_VALUE is generated if <index> is greater than or equal to the implementation-dependent number of supported program local parameters for the program target.

When the program target type is VERTEX_PROGRAM_ARB, each program local parameter returned is an array of four values.

The command

```
  void GetProgramivARB(enum target, enum pname, int *params);
```

obtains program state for the program target <target>, writing the state into the array given by <params>. GetProgramivARB can be used to determine the properties of the currently bound program object or implementation limits for <target>.

If <pname> is PROGRAM_LENGTH_ARB, PROGRAM_FORMAT_ARB, or PROGRAM_BINDING_ARB, GetProgramivARB returns one integer holding the program string length (in bytes), program string format, and program name, respectively, for the program object currently bound to <target>.

If <pname> is MAX_PROGRAM_LOCAL_PARAMETERS_ARB or MAX_PROGRAM_ENV_PARAMETERS_ARB, GetProgramivARB returns one integer holding the maximum number of program local parameters or program environment parameters, respectively, supported for the program target <target>.

If <pname> is MAX_PROGRAM_INSTRUCTIONS_ARB, MAX_PROGRAM_TEMPORARIES_ARB, MAX_PROGRAM_PARAMETERS_ARB, MAX_PROGRAM_ATTRIBS_ARB, or MAX_PROGRAM_ADDRESS_REGISTERS_ARB, GetProgramivARB returns a single integer giving the maximum number of instructions, temporaries, parameters, attributes, and address registers that can be used by a program of type <target>. If <pname> is PROGRAM_INSTRUCTIONS_ARB, PROGRAM_TEMPORARIES_ARB, PROGRAM_PARAMETERS_ARB, PROGRAM_ATTRIBS_ARB, or PROGRAM_ADDRESS_REGISTERS_ARB, GetProgramivARB returns a single integer giving the number of instructions, temporaries, parameters, attributes, and address registers used by the current program for <target>.

If <pname> is MAX_PROGRAM_NATIVE_INSTRUCTIONS_ARB, MAX_PROGRAM_NATIVE_TEMPORARIES_ARB, MAX_PROGRAM_NATIVE_PARAMETERS_ARB, MAX_PROGRAM_NATIVE_ATTRIBS_ARB, or MAX_PROGRAM_NATIVE_ADDRESS_REGISTERS_ARB, GetProgramivARB returns a single integer giving the maximum number of native instruction, temporary, parameter, attribute, and address register resources available to a program of type <target>. If <pname> is PROGRAM_NATIVE_INSTRUCTIONS_ARB, PROGRAM_NATIVE_TEMPORARIES_ARB, PROGRAM_NATIVE_PARAMETERS_ARB, PROGRAM_NATIVE_ATTRIBS_ARB, or PROGRAM_NATIVE_ADDRESS_REGISTERS_ARB, GetProgramivARB returns a single integer giving the number of native instruction, temporary, parameter, attribute, and address register resources consumed by the program currently bound to <target>. Native resource counts will reflect the results of implementation-dependent scheduling and optimization algorithms applied by the GL, as well as emulation of non-native features. If <pname> is PROGRAM_UNDER_NATIVE_LIMITS_ARB, GetProgramivARB returns 0 if the native resource consumption of the program currently bound to <target> exceeds the number of available resources for any resource type, and 1 otherwise.

The command

```
  void GetProgramStringARB(enum target, enum pname, void *string);
```

obtains the program string for the program object bound to <target> and places the information in the array <string>. <pname> must be PROGRAM_STRING_ARB. <n> ubytes are returned into the array program where <n> is the length of the program in ubytes, as returned by GetProgramivARB when <pname> is PROGRAM_LENGTH_ARB. The program string is always returned using the format given when the program string was specified.

The commands

```
  void GetVertexAttribdvARB(uint index, enum pname, double *params);
  void GetVertexAttribfvARB(uint index, enum pname, float *params);
  void GetVertexAttribivARB(uint index, enum pname, int *params);
```

obtain the vertex attribute state named by <pname> for the vertex attribute numbered <index> and places the information in the array <params>. <pname> must be one of VERTEX_ATTRIB_ARRAY_ENABLED_ARB, VERTEX_ATTRIB_ARRAY_SIZE_ARB, VERTEX_ATTRIB_ARRAY_STRIDE_ARB, VERTEX_ATTRIB_ARRAY_TYPE_ARB, VERTEX_ATTRIB_ARRAY_NORMALIZED_ARB, or CURRENT_VERTEX_ATTRIB_ARB. Note that all the queries except CURRENT_VERTEX_ATTRIB_ARB return client state. The error INVALID_VALUE is generated if <index> is greater than or equal to MAX_VERTEX_ATTRIBS_ARB. The error INVALID_OPERATION is generated if <index> is zero and <pname> is CURRENT_VERTEX_ATTRIB_ARB, as there is no current value for vertex attribute zero.

The command

```
  void GetVertexAttribPointervARB(uint index, enum pname, void **pointer);
```

obtains the pointer named <pname> for vertex attribute numbered <index> and places the information in the array <pointer>. <pname> must be VERTEX_ATTRIB_ARRAY_POINTER_ARB. The INVALID_VALUE error is generated if <index> is greater than or equal to MAX_VERTEX_ATTRIBS_ARB.

The command

```
  boolean IsProgramARB(uint program);
```

returns TRUE if <program> is the name of a program object. If <program> is zero or is a non-zero value that is not the name of a program object, or if an error condition occurs, IsProgramARB returns FALSE. A name returned by GenProgramsARB, but not yet bound, is not the name of a program object.

Concerning Section 6.1.12, Saving and Restoring State (p. 207):

(no actual modifications to the spec) Only the enables, current vertex attributes, and vertex array state introduced by this extension can be pushed and popped. See the attribute column in Table X.6 for determining what vertex program state can be pushed and popped with PushAttrib, PopAttrib, PushClientAttrib, and PopClientAttrib.

Additions to Appendix A of the OpenGL 1.3 Specification (Invariance)

Add to end of Section A.3 (p. 242):

Rule 4. Vertex program instructions not relevant to the calculation of any result must have no effect on that result.

Rule 5. Vertex program instructions relevant to the calculation of any result must always produce the identical result.

Instructions relevant to the calculation of a result are any instructions in a sequence of instructions that eventually determine the source values for the calculation under consideration.

There is no guaranteed invariance between vertices transformed by conventional GL vertex transform mode and vertices transformed by vertex program mode. Multi-pass rendering algorithms that require rendering invariances to operate correctly should not mix conventional GL vertex transform mode with vertex program mode for different rendering passes, except by using the position invariance option (section 2.14.4.5.1) in all vertex program mode passes. However, such algorithms will operate correctly if the algorithms limit themselves to a single mode of vertex transformation.

Additions to the AGL/GLX/WGL Specifications

Program objects are shared between AGL/GLX/WGL rendering contexts if and only if the rendering contexts share display lists. No change is made to the AGL/GLX/WGL API.

Changes to program objects shared between multiple rendering contexts will be serialized (i.e., the changes will occur in a specific order).

Changes to a program object made by one rendering context are not guaranteed to take effect in another rendering context until the other calls BindProgram to bind the program object.

When a program object is deleted by one rendering context, the object itself is not destroyed until it is no longer the current program object in any context. However, the name of the deleted object is removed from the program object name space, so the next attempt to bind a program using the same name will create a new program object. Recall that destroying a program object bound in the current rendering context effectively unbinds the object being destroyed.

Dependencies on EXT_vertex_weighting and ARB_vertex_blend

If EXT_vertex_weighting and ARB_vertex_blend are both not supported, all discussions of vertex weights should be removed.

In particular, references to vertex weights should be removed from Table X.1, and the description of ArrayElement in section 2.8. The line

```
  "weight" <vtxOptWeightNum>
```

should be removed from the <vtxAttribItem> grammar rule, and the grammar rules <vtxOptWeightNum> and <vtxWeightNum> should be deleted. "vertex.weight" and "vertex.weight[n]" should be removed from Table X.2. The discussion of vertex weights in section 2.14.3.1 should be removed.

Additionally, the first line of Table X.3.8 should be modified to read:

Binding	Underlying State
state.matrix.modelview	modelview matrix

Dependencies on ARB_matrix_palette:

If ARB_matrix_palette is not supported, all discussions of per-vertex matrix indices and the matrix palette should be removed.

In particular, the reference to matrix indices should be removed from the description of ArrayElement in section 2.8. The line

```
  "matrixindex" "[" <vtxWeightNum> "]"
```

should be removed from the <vtxAttribItem> grammar rule. The line

```
  "palette" "[" <statePaletteMatNum> "]"
```

should be removed from the <stateMatrixName> grammar rule, and the

<statePaletteMatNum> grammar rule should be removed entirely. "vertex.matrixindex[n]" should be removed from Table X.2, and "state.matrix.palette[n]" should be removed from Table X.3.8. The discussion of vertex matrix indices in section 2.14.3.1 should be removed.

Dependencies on EXT_point_parameters and ARB_point_parameters

The discussion of point size determination in EXT/ARB_point_parameters should qualified to indicate that this functionality only applies when vertex program mode is disabled.

If EXT/ARB_point_parameters is not supported, references to point parameter state should be eliminated. In particular,

"attenuation"

should be eliminated from the <statePointProperty> grammar rule, and the corresponding entries in Table X.3.7 should be eliminated.

Additionally, references to the minimum and maximum point sizes and the fade threshold should be removed from Table X.3.7 and the explanatory text immediately thereafter. The components column of the "state.point.size" binding in Table X.3.7 should read (s,0,0,1).

Even if EXT/ARB_point_parameters is not supported, the point size result (result.pointsize) still operates as specified.

Dependencies on EXT_fog_coord

If EXT_fog_coord is not supported, references to fog coordinates should be removed from Table X.1, and the description of ArrayElement in section 2.8. The line "fogcoord" should be removed from the <vtxAttribItem> grammar rule, and "vertex.fogcoord" should be removed from Table X.2. Also, the use of FOG_COORDINATE_SOURCE_EXT in section 3.10 should be removed.

Even if EXT_fog_coord is not supported, the fog coordinate output (result.fogcoord) still operates as specified. When in vertex program mode, there are no well-defined eye coordinates that could be used for fog. This means that the functionality of EXT_fog_coord is required to implement ARB_vertex_program even if the EXT_fog_coord extension itself is not supported.

Dependencies on EXT_secondary_color

If EXT_secondary_color is not supported, references to secondary color should be removed from Table X.1, and the description of ArrayElement in section 2.8. The line "secondary" should be removed from the <vtxOptColorType> grammar rule, and "vertex.color.secondary" should be removed from Table X.2.

Even if EXT_secondary_color is not supported, the secondary color results (result.color.secondary, result.color.front.secondary, result.color.back.secondary) still operate as specified in program mode, and when in program mode, the color sum enable behaves exactly as specified in EXT_secondary_color. These vertex result registers are required to implement OpenGL 1.2's separate specular mode within a vertex program.

The color sum enable enumerant from EXT_secondary_color has been brought over and renamed to COLOR_SUM_ARB. The enumerant value itself is unchanged from EXT_secondary_color.

Dependencies on ARB_transpose_matrix

If ARB_transpose_matrix is not supported, the discussion of TRANSPOSE_CURRENT_MATRIX_ARB in the edits to section 6.1.2 should be removed.

Interactions with NV_vertex_program

The existing NV_vertex_program extension, if supported, also provides a similar vertex programming model. This extension is incompatible with NV_vertex_program in a number of different ways. Mixing the two models in a single application is possible but not recommended. The interactions between the extensions are defined below.

Functions, enumerants, and programs defined in NV_vertex_program are called "NV functions", "NV enumerants", and "NV programs" respectively. Functions, enumerants, and programs defined in ARB_vertex_program are called "ARB functions", "ARB enumerants", and "ARB programs" respectively.

The following enumerants are identical in the two extensions:

ARB_vertex_program	NV_vertex_program
VERTEX_PROGRAM_ARB	VERTEX_PROGRAM_NV
VERTEX_PROGRAM_POINT_SIZE_ARB	VERTEX_PROGRAM_POINT_SIZE_NV
VERTEX_PROGRAM_TWO_SIDE_ARB	VERTEX_PROGRAM_TWO_SIDE_NV
VERTEX_ATTRIB_ARRAY_SIZE_ARB	ATTRIB_ARRAY_SIZE_NV
VERTEX_ATTRIB_ARRAY_STRIDE_ARB	ATTRIB_ARRAY_STRIDE_NV
VERTEX_ATTRIB_ARRAY_TYPE_ARB	ATTRIB_ARRAY_TYPE_NV
CURRENT_VERTEX_ATTRIB_ARB	CURRENT_ATTRIB_NV
VERTEX_ATTRIB_ARRAY_POINTER_ARB	ATTRIB_ARRAY_POINTER_NV
PROGRAM_LENGTH_ARB	PROGRAM_LENGTH_NV
PROGRAM_STRING_ARB	PROGRAM_STRING_NV
PROGRAM_ERROR_POSITION_ARB	PROGRAM_ERROR_POSITION_NV
CURRENT_MATRIX_ARB	CURRENT_MATRIX_NV
CURRENT_MATRIX_STACK_DEPTH_ARB	CURRENT_MATRIX_STACK_DEPTH_NV
MAX_PROGRAM_MATRICES_ARB	MAX_TRACK_MATRICES_NV
MAX_PROGRAM_MATRIX_STACK_DEPTH_ARB	MAX_TRACK_MATRIX_STACK_DEPTH_NV

The following GL state is identical in the two extensions and can be set or queried using either NV functions or ARB functions.

- Vertex program mode enable.

- Vertex program point size mode enable.

- Vertex program two sided mode enable.

- Program error position.

- NV_vertex_program "program parameters" and ARB_vertex_program "program environment parameters".

- Current values of generic vertex attributes. Conventional and generic vertex attributes will alias according to the NV_vertex_program spec, which is permissible but optional under ARB_vertex_program.

- NV_vertex_program "tracking matrices" and ARB_vertex_program "program matrices". The NV and ARB enumerants passed to MatrixMode are different, however.

- Vertex attribute array sizes, types, strides, and pointers.

- Vertex program object names, targets, formats, program string, program string lengths, and residency information. The ARB and NV query functions operate differently. The ARB query function does not allow queries of target (passed in to the query) and residency information. The NV query function does not allow queries of program name (passed in to the query) or format. The format of NV programs is always PROGRAM_FORMAT_ASCII_ARB.

- Current matrix and current matrix stack depth.

- Implementation-dependent limits on number of tracking/program matrices and tracking/program matrix stack depth.

- Program object name space. Program objects are created differently in the NV and ARB specs. Under the NV spec, program objects are created by calling LoadProgramNV. Under the ARB spec, program objects are created by calling BindProgramARB with an unused program name.

The following state is provided only by ARB_vertex_program:

- Program error string. Querying the error string after calling LoadProgramNV produces undefined results.

- Vertex attribute array normalization enables. Setting up vertex attribute arrays through NV functions will set the normalization enable appropriately based on the NV spec.

- Vertex program object resource counts and native resource counts. These values are undefined for NV programs.

- Vertex program local parameters. They can not be used by NV programs.

- Implementation-dependent limits on the number of program environment parameters, program local parameters, resource counts, and native resource counts. These limits are baked into the NV spec, except for native counts, which don't exist.

The following state is provided only by NV_vertex_program:

- TrackMatrix enables and transforms.

- Generic vertex attribute evaluator maps. The NV evaluator functionality will be supported even for ARB programs.

The following are additional functional differences between ARB_vertex_program and NV_vertex_program:

- ARB program temporaries, address registers, and result registers are initially undefined. The corresponding values in NV programs are initialized to (0,0,0,0), 0, and (0,0,0,1), respectively. ARB programs running on NV_vertex_program platforms can not rely on NV_vertex_program initialization behavior for temporaries or address registers, but result registers will be initialized to (0,0,0,1). In any event, ARB programs that rely on NV_vertex_program initialization may not behave properly on other platforms that support ARB_vertex_program but not NV_vertex_program.

- NV programs use a set of fixed variable and register names, with no support for user-defined variables. ARB programs provide no support for fixed variable names; all variables must be declared, explicitly or implicitly, in the program.

- ARB programs support parameter variables that can be bound to selected GL state variables, and are updated automatically when the underlying state changes. NV programs provide no such support; applications must set program parameters themselves.

- ARB programs allow program constants to be declared in the program text; NV programs require that constants be loaded into the program parameter array.

- ARB programs support program local parameters; NV programs do not. Applications using multiple NV programs must manage the shared program parameter array appropriately.

- ARB_vertex_program vertex array support provides a normalized flag to optionally normalize fixed-point array data to the [0,1] or [-1,1] range. ARB_vertex_program also provides several immediate-mode entry points with the same support. NV_vertex_program supports normalized data only for unsigned byte data types, and does not support non-normalized unsigned bytes. VertexAttrib4ub{v}NV was renamed to VertexAttrib4Nub{v}ARB to indicate that the 4ub call normalizes its parameters to a [0,1] range.

- ARB_vertex_blend and ARB_matrix_palette support are documented by the ARB spec, but not by the NV spec.

- ARB_vertex_program contains an OPTION mechanism for future extensibility, and a position invariant program option. Both features are found in NV_vertex_program1_1, but not in NV_vertex_program.

- NV_vertex_program supports a vertex state program target that allows programs to write to program parameters (VERTEX_STATE_PROGRAM_NV). No such support exists in ARB_vertex_program. Running a NV state program will update the program parameter/program environment parameter array, and such updates can be visible through ARB programs.

- LoadProgramNV entry point was changed to ProgramStringARB to match OpenGL convention that a verb should not be included in a command name that merely sets state.

- The formal parameter name for program objects was "id" in NV_vertex_program; in ARB_vertex_program, this formal name is now "program" to match how texture object routines name their formal texture object names "texture".

- NV_vertex_program has language that makes it sound that LoadProgramNV (ProgramStringARB) only accepts the VERTEX_PROGRAM_NV target and the start token must be "!!VP1.0". This extension clarifies the language so that it is clear that other targets and start token types are permitted.

- NV_vertex_program numeric requirements are not present in the ARB spec. The ARB spec requires nothing more than the numeric requirements spelled out in section 2.1.1 (Floating-Point Computations) in the core specification.

- ARB programs allow single instructions to source multiple distinct vertex attributes or program parameters. NV programs do not. On current NV_vertex_program hardware, such instructions may require additional instructions and temporaries to execute.

- ARB programs support the folowing instructions not supported by NV "VP1.0" programs:

  * ABS: absolute value. Supported on VP1.1 NV programs, but not on VP1.0 programs. Equivalent to "MAX dst, src, -src".

  * EX2: exponential base 2. On VP1.0 and VP1.1 hardware, this instruction will be emulated using EXP and a number of additional instructions.

  * FLR: floor. On VP1.0 and VP1.1 hardware, this instruction will be emulated using an EXP and an ADD instruction.

  * FRC: fraction. On VP1.0 and VP1.1 hardware, this instruction will be emulated using an EXP instruction, and possibly a MOV instruction to replicate the scalar result.

  * LG2: logarithm base 2. On VP1.0 and VP1.1 hardware, this instruction will be emulated using LOG and a number of additional instructions.

  * POW: exponentiation. On VP1.0 and VP1.1 hardware, this instruction will be emulated using LOG, MUL, and EXP instructions, and possibly additional instructions to generate a high-precision result.

  * SUB: subtraction. Supported on VP1.1 NV programs, but not on VP1.0 programs. Equivalent to "ADD dst, src1, -src2".

  * SWZ: extended swizzle. On VP1.0 and VP1.1 hardware, this instruction will be emulated using a single MAD instruction and a program parameter constant.

  * XPD: cross product. On VP1.0 and VP1.1 hardware, this instruction will be emulated using a MUL and a MAD instruction.

- The COLOR_SUM_EXT enable is ignored when NV programs are executed (default secondary color outputs are zero) but not when ARB programs are executed (default secondary color outputs are undefined). The driver will take care of the color sum operation based on which type of program is currently bound.

- NV programs are required to write a vertex position; ARB programs are not.

- There is both an ARB and an NV boolean enable for each generic array (two booleans per generic array). Each generic array's NV enable is enabled with EnableClientState(VERTEX_ATTRIB_ARRAYn_NV) or disabled with DisableClientState(VERTEX_ATTRIB_ARRAYn_NV) while each generic array's ARB enable is enabled with EnableVertexAttribArrayARB(n) and disabled with DisableVertexAttribArrayARB(n).

  Enabling (or disabling) an ARB generic array enables (or disables) BOTH the NV and ARB generic array booleans.

  However enabling (or disabling) the NV generic array enable changes only the NV generic array enable (the ARB enable is UNchanged).

  When an enabled valid current vertex program (whether specified as an ARB or NV vertex program) is bound, the NV generic array enables are considered (and the ARB enables are ignored). If a given NV generic array enable is true, the corresponding generic array state is applied. However if there is an enabled valid vertex program and a particular NV generic array is disabled, then the corresponding conventional aliased array state is applied.

  When the current vertex program is disabled or not valid (so conventional vertex processing is performed), the ARB generic array enables are considered (and the NV enables are ignored). If a given ARB generic array enable is true, the corresponding generic array state is applied. However if the current vertex program is disabled or NOT valid and a particular ARB generic array is disabled, then the corresponding conventional aliased array state is applied.

  This behavior means generic vertex arrays can be applied to conventional vertex processing when the ARB generic vertex array enable boolean is true. For example, you can send normalized UNSIGNED_SHORT texture coordinate set arrays as aliased generic vertex arrays where conventionally UNSIGNED_SHORT texture coordinate set arrays are unnormalized.

NV_vertex_program interaction Issues:

- Should matrix tracking support extend to ARB program environment parameters?

Interactions with EXT_vertex_shader

The existing EXT_vertex_shader extension, if supported, also provides a similar vertex programming model. This extension is incompatible with

ARB_vertex_program in a number of different ways. Mixing the two models in a single application is possible but not recommended. The interactions between the extensions are defined below.

First, it should be trivially noted that an EXT_vertex_shader "shader" serves the same purpose as an ARB_vertex_program "program". The two terms will be used interchangeably throughout this discussion.

The most obvious difference between the two extensions is that the definition of the vertex program is accomplished in EXT_vertex_shader through the use of instruction-specifying procedure calls and is accomplished in ARB_vertex_program by providing a textual string describing the program. This is mostly a distinction of interface rather than functionality.

Each extension provides its own distinct set of GL state, entry points, and enumerants. However, there are several areas of overlap both in conceptual framework and in programming model that are worth noting for those familiar with both API's.

1. Resource terminology and types

   Both ARB_vertex_program and EXT_vertex_shader offer access to similar types of resources for use by vertex programs.

   The following terms describe roughly equivalent resources in their respective extensions:

EXT_vertex_shader	ARB_vertex_program	Note
instructions	instructions	
variants	attributes	
locals	temporaries	
local constants	parameters bound to inline constants	(a)
invariants	parameters bound to GL state and program environment parameters	(b)

   a. ARB_vertex_program has no intrinsic storage type that corresponds to EXT_vertex_shader's LOCAL_CONSTANT storage type, but rather supports program parameters bound to inline constant vectors specified within the program text. This essentially makes LOCAL_CONSTANT a special case of an ARB_vertex_program program parameter. The values of these inline constant parameters can not be changed without redefining the program itself, just like the values of EXT_vertex_shader LOCAL_CONSTANTs.

   b. ARB_vertex_program has no intrinsic storage type that corresponds to EXT_vertex_shader's INVARIANT storage type, but rather supports program parameters bound to GL state variables, program environment parameters, and program local parameters. This essentially makes INVARIANT a special case of an ARB_vertex_program program parameter. The values of these bound program parameters can be changed without redefining the program itself, but remain constant from vertex to vertex during vertex program execution, just like the values of EXT_vertex_shader INVARIANTs.

```
    ARB_vertex_program also adds the concept of a program local parameter,
    which has no direct analogue in EXT_vertex_shader, as it represents a
    parameter that is stored locally with the program object, but the
    values of these parameters can be changed without redefining the
    program itself.

 2. Resource usage queries

    Both ARB_vertex_program and EXT_vertex_shader provide queries to assist
    in determining the resource usage of a given shader and whether the
    shader would "fit" within the limits imposed by the underlying hardware
    implementation. The application can investigate the maximum numbers of
    shader resources supported by an implementation, shader resources
    available in hardware, and resources consumed by a given shader after
    being compiled into the implementation's native representation.

    In EXT_vertex_shader (see the end of section 2.14 of the
    EXT_vertex_shader specification), the queries are handled by glGet.

    In ARB_vertex_programs (see section 2.14.3.7 of this specification),
    similar queries are handled by GetProgramivARB, with a target of
    VERTEX_PROGRAM_ARB.

    The following queries exist in both extensions and serve roughly
    equivalent purposes in each:
```

EXT_vertex_shader	ARB_vertex_program
MAX_VERTEX_SHADER_INSTRUCTIONS_EXT	MAX_PROGRAM_INSTRUCTIONS_ARB
MAX_VERTEX_SHADER_VARIANTS_EXT	MAX_PROGRAM_ATTRIBS_ARB
MAX_VERTEX_SHADER_LOCALS_EXT	MAX_PROGRAM_TEMPORARIES_ARB
MAX_OPTIMIZED_VERTEX_SHADER_INSTRUCTIONS_EXT	MAX_PROGRAM_NATIVE_INSTRUCTIONS_ARB
MAX_OPTIMIZED_VERTEX_SHADER_VARIANTS_EXT	MAX_PROGRAM_NATIVE_ATTRIBS_ARB
MAX_OPTIMIZED_VERTEX_SHADER_LOCALS_EXT	MAX_PROGRAM_NATIVE_TEMPORARIES_ARB
VERTEX_SHADER_INSTRUCTIONS_EXT	PROGRAM_NATIVE_INSTRUCTIONS_ARB
VERTEX_SHADER_VARIANTS_EXT	PROGRAM_NATIVE_ATTRIBS_ARB
VERTEX_SHADER_LOCALS_EXT	PROGRAM_NATIVE_TEMPORARIES_ARB
VERTEX_SHADER_OPTIMIZED_EXT	PROGRAM_UNDER_NATIVE_LIMITS_ARB

```
    ARB_vertex_program offers additional queries to account for differences
    in some of the resource definitions (program environment parameters and
    program local parameters, address registers, etc.) as well as the
    ability to separately query a compiled program's resource usage
    according to the specification versus a possibly more efficient
    resource usage obtained by passing the program through by a "smart"
    compiler.

    The following queries do not exist in ARB_vertex_program due to the
    slightly different resource models:
```

EXT_vertex_shader	ARB_vertex_program
{MAX_}{OPTIMIZED_}VERTEX_SHADER_INVARIANTS_EXT	(a)
{MAX_}{OPTIMIZED_}VERTEX_SHADER_LOCAL_CONSTANTS_EXT	(a)

```
      a. ARB_vertex_program coalesces all of the different program
         parameters (environment, local, inline constant, and those bound
```

to GL state) into a single queryable resource for PROGRAM_PARAMETERS. EXT_vertex_shader provides separate queries even though these parameters may consume the same resource on some implementations.

The following queries do not exist in EXT_vertex_shader due to the slightly different resource models:

EXT_vertex_shader	ARB_vertex_program
(b)	PROGRAM_*_ARB
(c)	{MAX_}{NATIVE_}PROGRAM_PARAMETERS_ARB
(d)	{MAX_}{NATIVE_}PROGRAM_ADDRESS_REGISTERS_ARB

b. These queries are used to find out how many resources a given program used according to the specification, *before* running the program through an optimizing compiler. This distinction is not made in EXT_vertex_shader.

c. These queries are used to find out how many parameters were used by a program or are allowed by an implementation, in total without distinguishing between environment parameters, program local parameters, inline constant parameters, or parameters bound to GL state. EXT_vertex_shader does not provide this information.

d. EXT_vertex_shader does not provide have any address register resources since all dynamic array references are handled with the atomic OP_INDEX instruction.

3. Symbols and variable names

In EXT_vertex_shader resources that represent storage locations (i.e. INVARIANTS, VARIANTS, LOCALS, LOCAL_CONSTANTS) are abstractly referenced through a GL-allocated symbol id obtained from GenSymbolsEXT. This level of abstraction is provided to allow the implementation to make hardware-dependent decisions about the best way to arrange, allocate, and re-use hardware resources.

Though ARB_vertex_program does not use symbol id's to refer to similar types of resources, it does provide similar functionality by allowing a vertex program to declare arbitrarily named variables for each resource in use. These names are assigned using the declaration syntax associated with the "PARAM", "ATTRIB", "TEMP", and "OUTPUT", and "ADDRESS" keywords.

4. Program management

With the exception of the actual program specification itself, EXT_vertex_shader and ARB_vertex_program have very similar program management API's.

The following procedures serve roughly equivalent functions in their respective extensions.

EXT_vertex_shader   ARB_vertex_program

-----------------	------------------
BindVertexShaderEXT	BindProgramARB
GenVertexShadersEXT	GenProgramsARB
DeleteVertexShaderEXT	DeleteProgramsARB

The following procedures are used in EXT_vertex_shader to define the program instruction sequence, and are not present in ARB_vertex_program since the string provided to ProgramStringARB fully defines the program contents.

```
    ShaderOp1EXT
    ShaderOp2EXT
    ShaderOp3EXT
    SwizzleEXT
    WriteMaskEXT
    InsertComponentEXT
    ExtractComponentEXT
```

5. Data specification routines

With the exception of the discrepancies in data types and resource names described above, EXT_vertex_shader and ARB_vertex_program provide similar program data specification and "current data query" API's.

The following procedures serve roughly equivalent functions in their respective extensions:

EXT_vertex_shader	ARB_vertex_program	Note
SetInvariantEXT	ProgramEnvParameter4*ARB	(a)
GetInvariant*vEXT	GetProgramEnvParameter*vARB	(a)
Variant*vEXT	VertexAttrib*ARB	
VariantPointerEXT	VertexAttribPointerARB	
GetVariant*vEXT	GetVertexAttrib*vARB	
GetVariantPointervEXT	GetVertexAttribPointervARB	
EnableVariantClientStateEXT	EnableVertexAttribArrayARB	
DisableVariantClientStateEXT	DisableVertexAttribArrayARB	
IsVariantEnabledEXT	GetVertexAttrib*vARB	(b)

a. See item #1 and #2 for more information on the relationship between EXT_vertex_shader invariants and ARB_vertex_program program parameters.

b. The enabled state of an attribute array in ARB_vertex_program can be queried with GetVertexAttrib*v and a parameter of VERTEX_ATTRIB_ARRAY_ENABLED_ARB. In EXT_vertex_shader there is a dedicated enabled query procedure.

However, there are some data specification routines in EXT_vertex_shader that have no procedure call analogue in ARB_vertex_program as their functions are subsumed by the string representation of the program itself.

The following procedures in EXT_vertex_shader have functionality roughly covered by the following strings within the program text in ARB_vertex_shader:

```
    EXT_vertex_shader            ARB_vertex_program           Note
    -----------------            ------------------           -----
    SetLocalConstantEXT          PARAM C = {<x>,<y>,<z>,<w>};  (c)

    BindLightParameterEXT        state.light[n].*
    BindMaterialParameterEXT     state.material.*              (d)
    BindTexGenParameterEXT       state.texgen[n].*

    BindTextureUnitParameterEXT
      CURRENT_TEXTURE_COORDS     vertex.texcoord[n]
      TEXTURE_MATRIX             state.matrix.texture[n]

    BindParameterEXT
      CURRENT_VERTEX_EXT         vertex.position
      CURRENT_NORMAL             vertex.normal
      CURRENT_COLOR              vertex.color.*
      MODELVIEW_MATRIX           state.matrix.modelview[n]
      PROJECTION_MATRIX          state.matrix.projection
      MVP_MATRIX_EXT             state.matrix.mvp
      COLOR_MATRIX               <unavailable>                 (e)
      CLIP_PLANE                 state.clip[n].plane
      FOG_COLOR                  state.fog.color
      FOG_DENSITY                state.fog.params.x
      FOG_START                  state.fog.params.y
      FOG_END                    state.fog.params.z
      LIGHT_MODEL_AMBIENT        state.lightmodel.ambient
```

c. Note that while EXT_vertex_shader style local constants can be specified using inline constants in the program text, there is no functionality in ARB_vertex_program that corresponds to the GetLocalConstant*vEXT call. That is, program parameters bound to inline constant vectors can be set in the text, but not queried from the application.

d. Note that while EXT_vertex_shader supports binding material properties to variants, ARB_vertex_shader only supports binding them to program parameters (invariants). See item #11 below for more information.

e. Note that while EXT_vertex_shader supports binding color matrix if the ARB_imaging subset is supported, ARB_vertex_shader does not allow for such a binding. See item #11 below for more information.

6. Data types

EXT_vertex_shader supports data types of SCALAR, VECTOR, and MATRIX.

ARB_vertex_program intrinsically supports only vectors, though it allows for the definition of a matrix as a contiguous allocation of four row vectors. Some operations that, in EXT_vertex_shader require scalar inputs or scalar outputs, will, in ARB_vertex_program, use the selected component of the source vector as input and/or replicate their output to all components .

Further, EXT_vertex_shader supports a pair of InsertComponents and ExtractComponents functions that are not available (nor required) in ARB_vertex_program, as they essentially provide for conversion between the SCALAR, VECTOR, and MATRIX data types.

7. Input swizzles and output write-masks

   In EXT_vertex_shader, write masks are specified as a type of "instruction", using WriteMaskEXT, while in ARB_vertex_program, write masks are specified as modifiers to the destination resource with writemask modifiers, such as ".xyz" or ".w".

   In EXT_vertex_shader, source operand swizzles (component re- ordering, negation, and hard-coding to the value 0 and +/- 1.0) are also specified as a type of "instruction", using SwizzleEXT.

   In ARB_vertex_program, swizzles can either be handled as instruction ("SWZ") or as part of a modifier of the source argument to an instruction. The only differences between the two methods is that the source modifiers in ARB_vertex_program do not provide the ability to use 0.0 and +/- 1.0, or negate individual components, while the "SWZ" instruction does.

8. Support for clipping and user clip planes

   Both extensions provide similar support for traditional clipping to the view frustum, namely that frustum clipping is not subsumed by vertex shader, or vertex program execution.

   Additionally, EXT_vertex_shader supports user clip planes by transforming the user clip planes from eye-space into clip space and clipping in the clip space coordinate system. This is supported as long as the projection matrix is non-singular.

   ARB_vertex_program provides similar functionality but only for programs specified using the "position invariant" option. For more information on user clip-plane support, see issue #20 and section 2.14.4.5.1 of this specification.

9. Support for glRasterPos

   EXT_vertex_shader does not support transforming the current raster position vertex by the current vertex shader, while ARB_vertex_program does.

10. Relative addressing.

   The string based syntax of ARB_vertex_program supports a relative addressing model where a given declared array can be dynamically dereferenced by first loading a declared ADDRESS register, using the "ARL" instruction with a value obtained at program execution then using that named ADDRESS register as the index to dereference a declared array of parameters later on. See section 2.14.3.5 of this specification for details.

   For example, in ARB_vertex_program you can specify the following piece of a program.

```
    PARAM arr[5]  = { program.env[0..4] };
    ADDRESS addr;
```

```
        ATTRIB v1 = vertex.attrib[1];
        ARL addr, v1;
        MOV result, arr[addr.x + 1];
```

    EXT_vertex_shader supports relative addressing as as a single atomic
    operation through the use of the instruction OP_INDEX_EXT, as in

```
        ShaderOp2EXT(OP_INDEX_EXT, <res>, <arg1>, <arg2>).
```

    OP_INDEX_EXT supports relative addressing by taking the value stored in
    the register referred to by <arg1> and adding that value to the register
    number referred to by <arg2>, and loading <res> with the value stored in
    the register at the resulting offset. EXT_vertex_shader has the
    requirement that the register referred to by <arg2> is allocated as one
    of a contiguous range of symbols obtained from a single call to
    GenSymbolsEXT.

    To achieve the same functionality as the above ARB_vertex_program, using
    EXT_vertex_shader, one could allocate a LOCAL symbol to hold a "fake"
    address register, and do a similar type of dynamic dereference
    operation, placing the output in a temporary LOCAL before giving it as
    an source argument to the "real" instruction.

```
     arr_contiguousArraySymbol =
          GenSymbolsEXT(GL_VECTOR_EXT, GL_LOCAL_EXT, GL_FULL_RANGE_EXT, 5);

     addr_fakeAddressRegSymbol =
          GenSymbolsEXT(GL_VECTOR_EXT, GL_LOCAL_EXT, GL_FULL_RANGE_EXT, 1);

     v1_srcSymbolForARLOp =
          GenSymbolsEXT(GL_VECTOR_EXT, GL_VARIANT_EXT, GL_FULL_RANGE_EXT, 1);

     temp =
          GenSymbolsEXT(GL_VECTOR_EXT, GL_LOCAL_EXT, GL_FULL_RANGE_EXT, 1);

     result_ForMovOpSymbol =
          GenSymbolsEXT(GL_VECTOR_EXT, GL_LOCAL_EXT, GL_FULL_RANGE_EXT, 1);

      // load fake ADDRESS register
      ExtractComponentEXT(
          addr_fakeAddressRegSymbol,
          v1_srcSymbolForARLOp,
          0);

      // do dynamic dereference into a temp
      ShaderOp2EXT(
          GL_OP_INDEX_EXT,
          temp,
          addr_fakeAddressRegSymbol,
          contiguousArraySymbol);

      // do operation we really wanted (MOV) using looked up src value
      ShaderOp1EXT(
          GL_OP_MOV_EXT,
          result_ForMovOpSymbol,
          temp,
          (arr_contiguousArraySymbol + 1));
```

11. Available GL state bindings

    Both EXT_vertex_shader and ARB_vertex_program offer the ability to bind program resources to pieces of OpenGL state so that the values of OpenGL state parameters are available to the program without the application having to copy the state values manually into program parameters.

    The two extensions differ in exactly which pieces of state are available to a vertex program, with the main difference being that ARB_vertex_program offers a more comprehensive set of state bindings.

    First, EXT_vertex_shader can bind pieces of GL state considered to be "scalar" values to a single SCALAR symbol, whereas ARB_vertex_program, which handles only vectors, packs up to 4 scalar bindings into a single vector parameter.

    Similarly, EXT_vertex_shader can bind pieces of GL state considered to be "matrix" values to a single MATRIX symbol, whereas ARB_vertex_program supports bindings to matrix data by using up to four vectors to store the rows of the matrix.

    Other differences between the state bindings available in both API's are listed below:

    a. In EXT_vertex_shader, the light attenuation factors (CONSTANT, LINEAR, QUADRATIC, and SPOT_EXPONENT), are available as separate SCALAR bindings.

       In ARB_vertex_program, the light attenuation factors are all packed into a single vector called state.light[n].attenuation with the CONSTANT, LINEAR, QUADRATIC, and SPOT_EXPONENT factors in the x,y,z, and w parameters respectively.

    b. In EXT_vertex_shader the spotlight direction (SPOT_DIRECTION) and spot light cutoff angle (SPOT_CUTOFF), are available as separate bindings.

       In ARB_vertex_program, these parameters are all packed into a single vector called state.light[n].spot.direction with the with the x,y,z parameters of the spotlight direction and the the *cosine* of the cutoff angle in the x,y,z, and w parameters respectively.

    c. In EXT_vertex_shader, the fog equation factors (FOG_DENSITY, FOG_START, FOG_END), are avaiable as separate SCALAR bindings.

       In ARB_vertex_program, the fog equation factors are all packed into a single vector called state.fog.params with the fog density, linear start, linear end, and pre-computed 1.0/ (end-start) factors in the x,y,z, and w parameters respectively.

    d. In EXT_vertex_shader, material properties can be bound to a variant (i.e. "attribute" in ARB_vertex_program terminology) and can change per vertex, and the changes take effect immediately.

In ARB_vertex_program, material properties can only be bound to program parameters, and any changes to material properties between a Begin/End pair are not guaranteed to take effect until the following End command.

e. In EXT_vertex_shader, the material shininess property is bound to a SCALAR variable.

   In ARB_vertex_program, the material shininess property is bound to a vector with elements { s, 0.0, 0.0, 1.0 } where "s" is the material shininess property.

f. In EXT_vertex_shader, a program can bind to the current modelview, projection, composite modelview-projection, color, and texture matrices only in their entirety.

   In ARB_vertex_program, a program can bind to individual rows of any matrix, with the exception of the color matrix, which is not available in ARB_vertex_program.

   Additionally, ARB_vertex_program adds the ability to bind to multiple modelview matrices, multiple palette matrices, and a set of matrices dedicated for use with vertex programs called "program matrices". Further, ARB_vertex_program offers the ability to bind to the inverse, transpose, and inverse_transpose of any of the matrices available for binding.

   If an application desires the functionality of binding to the color matrix in ARB_vertex_program, that application can use one of the other matrices, for instance program matrices, to store the current color matrix.

12. Instruction set differences.

In general, ARB_vertex_program's instruction set is a super-set of the EXT_vertex_shader instructions that take VECTOR inputs and produce VECTOR outputs. The versions of the EXT_vertex_shader instructions that take non-vector (i.e. SCALAR or MATRIX) operands are almost all available in vector form as well.

The instructions from each set correspond as follows:

EXT_vertex_shader	ARB_vertex_program	Note
OP_INDEX_EXT	<unavailable>	(a)
OP_NEGATE_EXT	<unavailable>	(b)
OP_DOT3_EXT	"DP3"	(c)
OP_DOT4_EXT	"DP4"	
OP_MUL_EXT	"MUL"	
OP_ADD_EXT	"ADD"	
OP_MADD_EXT	"MAD"	
OP_FRAC_EXT	"FRC"	
OP_MAX_EXT	"MAX"	
OP_MIN_EXT	"MIN"	
OP_SET_GE_EXT	"SGE"	
OP_SET_LT_EXT	"SLT"	
OP_CLAMP_EXT	<unavailable>	(d)

```
 OP_FLOOR_EXT            "FLR"
 OP_ROUND_EXT           <unavailable>         (e)
 OP_EXP_BASE_2_EXT       "EX2"                (f)
 OP_LOG_BASE_2_EXT       "LG2"                (g)
 OP_POWER_EXT            "POW"                (h)
 OP_RECIP_EXT            "RCP"                (i)
 OP_RECIP_SQRT_EXT       "RSQ"                (j)
 OP_SUB_EXT              "SUB"
 OP_CROSS_PRODUCT_EXT    "XPD"                (k)
 OP_MULTIPLY_MATRIX_EXT <unavailable>         (l)
 OP_MOV_EXT              "MOV"
<unavailable>            "ARL"                (a)
<unavailable>            "ABS"
<unavailable>            "LIT"
<unavailable>            "EXP"                (f)
<unavailable>            "LOG"                (g)
<unavailable>            "DPH"
<unavailable>            "DST"
```

There are a few minor differences, however.

a. EXT_vertex_shader's OP_INDEX_EXT is not available in ARB_vertex_program which uses the "ARL" instruction and array syntax to handle dynamically dereferencing source data. See item #10 above and the discussion of "ARL" in section 2.14.3.5.

b. EXT_vertex_shader's OP_NEGATE_EXT is not available in ARB_vertex_program. ARB_vertex_program can support a "NEGATE" operation through the use of swizzle modifiers on source operands or the "SWZ" instruction.

```
  MOV tempA, -tempB;
```

or

```
  SWZ tempA, -tempB, x,y,z,w;
```

c. The "w" component of EXT_vertex_shader's OP_DOT3_EXT instruction is left unchanged.

However, in ARB_vertex_program, the "w" component gets the same result as the "x", "y", and "z" components.

d. EXT_vertex_shader's OP_CLAMP_EXT is not available in ARB_vertex_program. ARB_vertex_program can support a "CLAMP" operation by using a pair of "MAX" and "MIN" instructions as in:

```
  # CLAMP arg1 to be within [arg2, arg3]
  MAX temp,   arg1, arg2;
  MIN result, temp, arg3;
```

e. EXT_vertex_shader's OP_ROUND_EXT is not available in ARB_vertex_program. ARB_vertex_program can support a "ROUND" operation by using a pair of "ADD" and "FLOOR" instructions as in:

```
  ADD   temp,   arg1, 0.5;
  FLOOR result, temp;
```

f. EXT_vertex_shader's OP_EXP_BASE_2_EXT is designed to support high precision calculations of base-2 exponentiation.

   ARB_vertex_program's "EX2" is the equivalent function, however ARB_vertex_program also offers an "EXP" function that is designed to support a lower precision approximation of base-2 exponentiation that can be further refined through an iterative process.

   On some implementations, both "EX2" and "EXP" may be carried out with the same high precision at no cost relative to each other. As such, if a vertex program is using "EXP" with the intent of iteratively refining the approximation by using several successive instructions it may be more efficient to use a single call to "EX2" and get the high precision with a single instruction.

   If on the other hand, a single approximation is good enough, there is no additional cost to using "EXP" on such implementations.

   Further note that in EXT_vertex_shader, OP_EXP_BASE_2_EXT is specified to take a scalar operand, whereas ARB_vertex_program's "EXP" and "EX2" instruction each take a vector operand, use the "x" component, and then write (partial) results to all components of a destination vector.

g. EXT_vertex_shader's OP_LOG_BASE_2_EXT is designed to support high precision calculations of base-2 logarithms.

   ARB_vertex_program's "LG2" is the equivalent function, however ARB_vertex_program also offers an "LOG" function that is designed to support a lower precision approximation of base-2 logarithms that can be further refined through an iterative process.

   On some implementations, both "LG2" and "LOG" may be carried out with the same high precision at no cost relative to each other. As such, if a vertex program is using "LOG" with the intent of iteratively refining the approximation by using several successive instructions it may be more efficient to use a single call to "LG2" and get the high precision with a single instruction.

   If on the other hand, a single approximation is good enough, there is no additional cost to using "LOG" on such implementations.

   Further note that in EXT_vertex_shader, OP_LOG_BASE_2_EXT is specified to take a scalar operand, whereas ARB_vertex_program's "LOG" and "LOG2" instruction each take a vector operand, use the "x" component, and then write (partial) results to all components of a destination vector.

h. EXT_vertex_shader's OP_POWER_EXT is designed to support high precision calculations of the power function.

   ARB_vertex_program's "POW" is the equivalent function.

   Further note that in EXT_vertex_shader, OP_POWER_EXT is specified to take a scalar operand, whereas ARB_vertex_program's "POW" instruction takes a vector operand, uses the "x" component, and replicates the same result to all components of a destination vector.

i. EXT_vertex_shader's OP_RECIP_EXT is specified to take a scalar operand, whereas ARB_vertex_program's "RCP" instruction takes a single component of a vector and replicates the same result to all components of the destination vector.

j. EXT_vertex_shader's OP_RECIP_SQRT_EXT is specified to take a scalar operand, whereas ARB_vertex_program's "RSQ" instruction takes a single component of a vector and replicates the same result to all components of the destination vector.

k. The "w" component of EXT_vertex_shader's OP_CROSS_PRODUCT_EXT instruction is forced to 1.0;

However, in ARB_vertex_program, the "w" component is left undefined and "writes to the w component of the destination are treated as disabled, regardless of the write mask specified in the XPD instruction".

l. EXT_vertex_shader's OP_MULTIPLY_MATRIX is not available in ARB_vertex_program. ARB_vertex_program can support a "MATRIX MULTIPLY" operation by using a series of "DP4" instructions as in:

```
PARAM mat[4] = { state.matrix.modelview };
DP4 result.x, vec, mat[0];
DP4 result.y, vec, mat[1];
DP4 result.z, vec, mat[2];
DP4 result.w, vec, mat[3];
```

13. Vertex provoking behavior

EXT_vertex_shader does not provoke vertex shader execution when variant 0 is specified (either using Variant*EXT, or variant arrays). Applications are required to use the conventional Vertex* or vertex arrays to provoke a vertex in both vertex shader mode and conventional mode. Variant 0 is considered current state and is queryable.

Conversely, ARB_vertex_program does provoke vertex program execution when attribute 0 is specified (either using VertexAttrib*vARB, or attribute arrays) in both vertex program mode and conventional mode. Attribute 0 is not considered current state and is not queryable.

For implementations that support both extensions, this means that if ARB_vertex_program is disabled, and EXT_vertex_shader is enabled, then specifying ARB_vertex_program's attribute 0 will still provoke execution of the currently bound EXT_vertex_shader defined shader.

14. Enabled state

On implementations that support both EXT_vertex_shader, and ARB_vertex_program, priority is given to ARB_vertex_program. That is to say, if both are enabled, the implementation uses the program defined by ARB_vertex_program and does not execute the currently bound EXT_vertex_shader shader unless or until ARB_vertex_program is subsequently disabled. Needless to say, it is not expected that a given application will actually attempt to use both vertex program API's at once.

GLX Protocol

The following rendering commands are sent to the server as part of a glXRender request:

```
    VertexAttrib1svARB
        2           12              rendering command length
        2           4189            rendering command opcode
        4           CARD32          index
        2           INT16           v[0]
        2                           unused

    VertexAttrib1fvARB
        2           12              rendering command length
        2           4193            rendering command opcode
        4           CARD32          index
        4           FLOAT32         v[0]

    VertexAttrib1dvARB
        2           16              rendering command length
        2           4197            rendering command opcode
        4           CARD32          index
        8           FLOAT64         v[0]

    VertexAttrib2svARB
        2           12              rendering command length
        2           4190            rendering command opcode
        4           CARD32          index
        2           INT16           v[0]
        2           INT16           v[1]

    VertexAttrib2fvARB
        2           16              rendering command length
        2           4194            rendering command opcode
        4           CARD32          index
        4           FLOAT32         v[0]
        4           FLOAT32         v[1]

    VertexAttrib2dvARB
        2           24              rendering command length
        2           4198            rendering command opcode
        4           CARD32          index
        8           FLOAT64         v[0]
        8           FLOAT64         v[1]

    VertexAttrib3svARB
        2           16              rendering command length
        2           4191            rendering command opcode
        4           CARD32          index
        2           INT16           v[0]
        2           INT16           v[1]
        2           INT16           v[2]
        2                           unused
```

```
VertexAttrib3fvARB
    2           20              rendering command length
    2           4195            rendering command opcode
    4           CARD32          index
    4           FLOAT32         v[0]
    4           FLOAT32         v[1]
    4           FLOAT32         v[2]

VertexAttrib3dvARB
    2           32              rendering command length
    2           4199            rendering command opcode
    4           CARD32          index
    8           FLOAT64         v[0]
    8           FLOAT64         v[1]
    8           FLOAT64         v[2]

VertexAttrib4bvARB
    2           12              rendering command length
    2           4230            rendering command opcode
    4           CARD32          index
    1           INT8            v[0]
    1           INT8            v[1]
    1           INT8            v[2]
    1           INT8            v[3]

VertexAttrib4svARB
    2           16              rendering command length
    2           4192            rendering command opcode
    4           CARD32          index
    2           INT16           v[0]
    2           INT16           v[1]
    2           INT16           v[2]
    2           INT16           v[3]

VertexAttrib4ivARB
    2           24              rendering command length
    2           4231            rendering command opcode
    4           CARD32          index
    4           INT32           v[0]
    4           INT32           v[1]
    4           INT32           v[2]
    4           INT32           v[3]

VertexAttrib4ubvARB
    2           12              rendering command length
    2           4232            rendering command opcode
    4           CARD32          index
    1           CARD8           v[0]
    1           CARD8           v[1]
    1           CARD8           v[2]
    1           CARD8           v[3]

VertexAttrib4usvARB
    2           16              rendering command length
    2           4233            rendering command opcode
    4           CARD32          index
```

```
           2           CARD16          v[0]
           2           CARD16          v[1]
           2           CARD16          v[2]
           2           CARD16          v[3]
```

VertexAttrib4uivARB
```
           2           24              rendering command length
           2           4234            rendering command opcode
           4           CARD32          index
           4           CARD32          v[0]
           4           CARD32          v[1]
           4           CARD32          v[2]
           4           CARD32          v[3]
```

VertexAttrib4fvARB
```
           2           24              rendering command length
           2           4196            rendering command opcode
           4           CARD32          index
           4           FLOAT32         v[0]
           4           FLOAT32         v[1]
           4           FLOAT32         v[2]
           4           FLOAT32         v[3]
```

VertexAttrib4dvARB
```
           2           40              rendering command length
           2           4200            rendering command opcode
           4           CARD32          index
           8           FLOAT64         v[0]
           8           FLOAT64         v[1]
           8           FLOAT64         v[2]
           8           FLOAT64         v[3]
```

VertexAttrib4NbvARB
```
           2           12              rendering command length
           2           4235            rendering command opcode
           4           CARD32          index
           1           INT8            v[0]
           1           INT8            v[1]
           1           INT8            v[2]
           1           INT8            v[3]
```

VertexAttrib4NsvARB
```
           2           16              rendering command length
           2           4236            rendering command opcode
           4           CARD32          index
           2           INT16           v[0]
           2           INT16           v[1]
           2           INT16           v[2]
           2           INT16           v[3]
```

VertexAttrib4NivARB
```
           2           24              rendering command length
           2           4237            rendering command opcode
           4           CARD32          index
           4           INT32           v[0]
           4           INT32           v[1]
           4           INT32           v[2]
```

```
        4           INT32           v[3]
```

VertexAttrib4NubvARB

```
        2           12              rendering command length
        2           4201            rendering command opcode
        4           CARD32          index
        1           CARD8           v[0]
        1           CARD8           v[1]
        1           CARD8           v[2]
        1           CARD8           v[3]
```

VertexAttrib4NusvARB

```
        2           16              rendering command length
        2           4238            rendering command opcode
        4           CARD32          index
        2           CARD16          v[0]
        2           CARD16          v[1]
        2           CARD16          v[2]
        2           CARD16          v[3]
```

VertexAttrib4NuivARB

```
        2           24              rendering command length
        2           4239            rendering command opcode
        4           CARD32          index
        4           CARD32          v[0]
        4           CARD32          v[1]
        4           CARD32          v[2]
        4           CARD32          v[3]
```

BindProgramARB

```
        2           12              rendering command length
        2           4180            rendering command opcode
        4           ENUM            target
        4           CARD32          program
```

ProgramEnvParameter4fvARB

```
        2           32              rendering command length
        2           4184            rendering command opcode
        4           ENUM            target
        4           CARD32          index
        4           FLOAT32         params[0]
        4           FLOAT32         params[1]
        4           FLOAT32         params[2]
        4           FLOAT32         params[3]
```

ProgramEnvParameter4dvARB

```
        2           44              rendering command length
        2           4185            rendering command opcode
        4           ENUM            target
        4           CARD32          index
        8           FLOAT64         params[0]
        8           FLOAT64         params[1]
        8           FLOAT64         params[2]
        8           FLOAT64         params[3]
```

ProgramLocalParameter4fvARB

```
        2           32              rendering command length
        2           4215            rendering command opcode
        4           ENUM            target
        4           CARD32          index
        4           FLOAT32         params[0]
        4           FLOAT32         params[1]
        4           FLOAT32         params[2]
        4           FLOAT32         params[3]
```

ProgramLocalParameter4dvARB

```
        2           44              rendering command length
        2           4216            rendering command opcode
        4           ENUM            target
        4           CARD32          index
        8           FLOAT64         params[0]
        8           FLOAT64         params[1]
        8           FLOAT64         params[2]
        8           FLOAT64         params[3]
```

The ProgramStringARB is potentially large, and hence can be sent in a glXRender or glXRenderLarge request.

ProgramStringARB

```
        2           16+len+p        rendering command length
        2           4217            rendering command opcode
        4           ENUM            target
        4           ENUM            format
        4           sizei           len
        len         LISTofBYTE      program
        p                           unused, p=pad(len)
```

If the command is encoded in a glxRenderLarge request, the command opcode and command length fields above are expanded to 4 bytes each:

```
        4           16+len+p        rendering command length
        4           4217            rendering command opcode
```

VertexAttribPointerARB, EnableVertexAttribArrayARB, and DisableVertexAttribArrayARB are entirely client-side commands.

The remaining commands are non-rendering commands. These commands are sent separately (i.e., not as part of a glXRender or glXRenderLarge request), using the glXVendorPrivateWithReply request:

DeleteProgramsARB

```
        1           CARD8           opcode (X assigned)
        1           17              GLX opcode (glXVendorPrivateWithReply)
        2           4+n             request length
        4           1294            vendor specific opcode
        4           GLX_CONTEXT_TAG context tag
        4           INT32           n
        n*4         LISTofCARD32    programs
```

```
GenProgramsARB
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           4               request length
    4           1295            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           INT32           n
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           n               reply length
    24                          unused
    n*4         LISTofCARD322   programs

GetProgramEnvParameterfvARB
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           6               request length
    4           1296            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           ENUM            target
    4           CARD32          index
    4                           unused
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           m               reply length, m=(n==1?0:n)
    4                           unused
    4           CARD32          n (number of parameter components)

    if (n=1) this follows:

    4           FLOAT32         params
    12                          unused

    otherwise this follows:

    16                          unused
    n*4         LISTofFLOAT32   params
```

```
        **GetProgramEnvParameterdvARB**
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           6               request length
            4           1297            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           ENUM            target
            4           CARD32          index
            4                           unused
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           m               reply length, m=(n==1?0:n*2)
            4                           unused
            4           CARD32          n (number of parameter components)

            if (n=1) this follows:

            8           FLOAT64         params
            8                           unused

            otherwise this follows:

            16                          unused
            n*8         LISTofFLOAT64   params

        **GetProgramLocalParameterfvARB**
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           6               request length
            4           1305            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           ENUM            target
            4           CARD32          index
            4           ENUM            pname
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           m               reply length, m=(n==1?0:n)
            4                           unused
            4           CARD32          n (number of parameter components)

            if (n=1) this follows:

            4           FLOAT32         params
            12                          unused

            otherwise this follows:

            16                          unused
            n*4         LISTofFLOAT32   params
```

GetProgramLocalParameterdvARB
```
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           6               request length
    4           1306            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           ENUM            target
    4           CARD32          index
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           m               reply length, m=(n==1?0:n*2)
    4                           unused
    4           CARD32          n (number of parameter components)

    if (n=1) this follows:

    8           FLOAT64         params
    8                           unused

    otherwise this follows:

    16                          unused
    n*8         LISTofFLOAT64   params
```

GetProgramivARB
```
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           5               request length
    4           1307            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           ENUM            target
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           m               reply length, m=(n==1?0:n)
    4                           unused
    4           CARD32          n

    if (n=1) this follows:

    4           INT32           params
    12                          unused

    otherwise this follows:

    16                          unused
    n*4         LISTofINT32     params
```

```
        GetProgramStringARB
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           5               request length
            4           1308            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           ENUM            target
            4           ENUM            pname
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           (n+p)/4         reply length
            4                           unused
            4           CARD32          n
            16                          unused
            n           STRING          program
            p                           unused, p=pad(n)

        GetVertexAttribdvARB
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           5               request length
            4           1301            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           INT32           index
            4           ENUM            pname
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           m               reply length, m=(n==1?0:n*2)
            4                           unused
            4           CARD32          n

            if (n=1) this follows:

            8           FLOAT64         params
            8                           unused

            otherwise this follows:

            16                          unused
            n*8         LISTofFLOAT64   params
        GetVertexAttribfvARB
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           5               request length
            4           1302            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           INT32           index
            4           ENUM            pname
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           m               reply length, m=(n==1?0:n)
            4                           unused
            4           CARD32          n

            if (n=1) this follows:

            4           FLOAT32         params
            12                          unused

            otherwise this follows:

            16                          unused
            n*4         LISTofFLOAT32   params

        GetVertexAttribivARB
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           5               request length
            4           1303            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           INT32           index
            4           ENUM            pname
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           m               reply length, m=(n==1?0:n)
            4                           unused
            4           CARD32          n

            if (n=1) this follows:

            4           INT32           params
            12                          unused

            otherwise this follows:

            16                          unused
            n*4         LISTofINT32     params
```

```
    IsProgramARB
        1           CARD8           opcode (X assigned)
        1           17              GLX opcode (glXVendorPrivateWithReply)
        2           4               request length
        4           1304            vendor specific opcode
        4           GLX_CONTEXT_TAG context tag
        4           INT32           n
      =>
        1           1               reply
        1                           unused
        2           CARD16          sequence number
        4           0               reply length
        4           BOOL32          return value
        20                          unused
```

    When transferring vertex attribute array elements, there may not be a
    protocol encoding that exactly matches the combination of combination of
    size, normalization enable, and data type in the array. If no match
    protocol encoding exists, the encoding for the corresponding 4-component
    attribute is used. v[1] and v[2] are set to zero if not specified in the
    vertex array. If v[3] is not specified in the vertex array, it is set to
    0x7F, 0x7FFF, 0x7FFFFFFF, 0xFF, 0xFFFF, or 0xFFFFFFFF for the
    VertexAttrib4NbvARB, VertexAttrib4NsvARB, VertexAttrib4NivARB,
    VertexAttrib4NubvARB, VertexAttrib4NusvARB, and VertexAttrib4NuivARB
    protocol encodings, respectively. v[3] is set to one if it is not
    specified in the vertex array for the the VertexAttrib4bvARB,
    VertexAttrib4svARB, VertexAttrib4ivARB, VertexAttrib4ubvARB,
    VertexAttrib4usvARB, and VertexAttrib4uivARB protocol encodings.

Errors

The error INVALID_VALUE is generated by any VertexAttrib*ARB or GetVertexAttrib*ARB command if <index> is greater than or equal to MAX_VERTEX_ATTRIBS_ARB.

The error INVALID_VALUE is generated by VertexAttribPointerARB or GetVertexAttribPointervARB if <index> is greater than or equal to MAX_VERTEX_ATTRIBS_ARB.

The error INVALID_VALUE is generated by VertexAttribPointerARB if <size> is not one of 1, 2, 3, or 4.

The error INVALID_VALUE is generated by VertexAttribPointerARB if <stride> is negative.

The error INVALID_VALUE is generated by EnableVertexAttribArrayARB or DisableVertexAttribArrayARB if <index> is greater than or equal to MAX_VERTEX_ATTRIBS_ARB.

The error INVALID_OPERATION is generated by ProgramStringARB if the program string <string> is syntactically incorrect or violates any semantic restriction of the execution environment of the specified program target <target>.

The error INVALID_OPERATION is generated by BindProgramARB if <program> is the name of a program whose target does not match <target>.

The error INVALID_VALUE is generated by any ProgramEnvParameter*ARB or GetProgramEnvParameter*ARB command if <index> is greater than or equal to the value of MAX_PROGRAM_ENV_PARAMETERS_ARB corresponding to the program target <target>.

The error INVALID_VALUE is generated by any ProgramLocalParameter*ARB or GetProgramLocalParameter*ARB command if <index> is greater than or equal to the value of MAX_PROGRAM_LOCAL_PARAMETERS_ARB corresponding to the program target <target>.

The error INVALID_OPERATION is generated if Begin, RasterPos, or any command that performs an explicit Begin is called when vertex program mode is enabled and the currently bound vertex program object does not contain a valid vertex program.

The error INVALID_OPERATION is generated by GetVertexAttrib*ARB if <index> is zero and <pname> is CURRENT_VERTEX_ATTRIB_ARB.

New State

Get Value	Type	Get Command	Initial Value	Description	Section	Attribute
VERTEX_PROGRAM_ARB	B	IsEnabled	False	vertex program enable	2.10	enable
VERTEX_PROGRAM_POINT_SIZE_ARB	B	IsEnabled	False	program-specified point size mode	2.14.3.7	enable
VERTEX_PROGRAM_TWO_SIDE_ARB	B	IsEnabled	False	two-sided color mode	2.14.3.7	enable
-	96+xR4	GetProgramEnv-ParameterARB	(0,0,0,0)	program environment parameters	2.14.1	-
CURRENT_VERTEX_ATTRIB_ARB	16+xR4	GetVertex-AttribARB	undefined	generic vertex attributes	2.7	current
PROGRAM_ERROR_POSITION_ARB	Z	GetIntegerv	-1	last program error position	2.14.1	-
PROGRAM_ERROR_STRING_ARB	0+xub	GetString	""	last program error string	2.14.1	-

Table X.6. New Accessible State Introduced by ARB_vertex_program.

Get Value	Type	Get Command	Initial Value	Description	Section	Attribute
VERTEX_ATTRIB_ARRAY_ENABLED_ARB	16+xB	GetVertex-AttribARB	False	vertex attrib array enable	2.8	vertex-array
VERTEX_ATTRIB_ARRAY_SIZE_ARB	16+xZ	GetVertex-AttribARB	4	vertex attrib array size	2.8	vertex-array
VERTEX_ATTRIB_ARRAY_STRIDE_ARB	16+xZ+	GetVertex-AttribARB	0	vertex attrib array stride	2.8	vertex-array
VERTEX_ATTRIB_ARRAY_TYPE_ARB	16+xZ4	GetVertex-AttribARB	FLOAT	vertex attrib array type	2.8	vertex-array
VERTEX_ATTRIB_ARRAY_ NORMALIZED_ARB	16+xB	GetVertex-AttribARB	False	vertex attrib array normalized	2.8	vertex-array
VERTEX_ATTRIB_ARRAY_POINTER_ARB	16+xP	GetVertex-AttribPointerARB	NULL	vertex attrib array pointer	2.8	vertex-array

Table X.7. New Accessible Client State Introduced by ARB_vertex_program.

Get Value	Type	Get Command	Initial Value	Description	Sec	Attrib
PROGRAM_BINDING_ARB	Z+	GetProgramivARB	object-specific	bound program name	6.1.12	-
PROGRAM_LENGTH_ARB	Z+	GetProgramivARB	0	bound program length	6.1.12	-
PROGRAM_FORMAT_ARB	Z1	GetProgramivARB	PROGRAM_FORMAT_ ASCII_ARB	bound program format	6.1.12	-
PROGRAM_STRING_ARB	ubxn	GetProgramStringARB	(empty)	bound program string	6.1.12	-
PROGRAM_INSTRUCTIONS_ARB	Z+	GetProgramivARB	0	bound program instructions	6.1.12	-
PROGRAM_TEMPORARIES_ARB	Z+	GetProgramivARB	0	bound program temporaries	6.1.12	-
PROGRAM_PARAMETERS_ARB	Z+	GetProgramivARB	0	bound program parameter bindings	6.1.12	-
PROGRAM_ATTRIBS_ARB	Z+	GetProgramivARB	0	bound program attribute bindings	6.1.12	-
PROGRAM_ADDRESS_REGISTERS_ARB	Z+	GetProgramivARB	0	bound program address registers	6.1.12	-
PROGRAM_NATIVE_INSTRUCTIONS_ARB	Z+	GetProgramivARB	0	bound program native instructions	6.1.12	-
PROGRAM_NATIVE_TEMPORARIES_ARB	Z+	GetProgramivARB	0	bound program native temporaries	6.1.12	-
PROGRAM_NATIVE_PARAMETERS_ARB	Z+	GetProgramivARB	0	bound program native parameter bindings	6.1.12	-
PROGRAM_NATIVE_ATTRIBS_ARB	Z+	GetProgramivARB	0	bound program native attribute bindings	6.1.12	-
PROGRAM_NATIVE_ADDRESS_ REGISTERS_ARB	Z+	GetProgramivARB	0	bound program native address registers	6.1.12	-
PROGRAM_UNDER_NATIVE_LIMITS_ARB	B	GetProgramivARB	0	bound program under native resource limits	6.1.12	-
-	96+xR4	GetProgramLocal- ParameterARB	(0,0,0,0)	bound program local parameter value	2.14.1	-

Table X.8. Program Object State. Program object queries return attributes of the program object currently bound to the program target <target>.

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
-	12+xR4	-	undefined	temporary registers	2.14.3.6	-
-	8+xR4	-	undefined	vertex result registers	2.14.3.7	-
	1+xZ1	-	undefined	vertex program address registers	2.14.3.8	-

Table X.9. Vertex Program Per-vertex Execution State. All per-vertex execution state registers are uninitialized at the beginning of program execution.

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
CURRENT_MATRIX_ARB	m*n*xM^4	GetFloatv	Identity	current matrix	6.1.2	-
CURRENT_MATRIX_STACK_DEPTH_ARB	m*Z+	GetIntegerv	1	current stack depth	6.1.2	-

Table X.10. Current matrix state where m is the total number of matrices including texture matrices and program matrices and n is the number of matrices on each particular matrix stack. Note that this state is aliased with existing matrix state.

New Implementation Dependent State

Get Value	Type	Get Command	Minimum Value	Description	Sec	Attrib
MAX_PROGRAM_ENV_PARAMETERS_ARB	Z+	GetProgramivARB	96	maximum program env parameters	2.14.1	-
MAX_PROGRAM_LOCAL_PARAMETERS_ARB	Z+	GetProgramivARB	96	maximum program local parameters	2.14.1	-
MAX_PROGRAM_MATRICES_ARB	Z+	GetIntegerv	8 (not to exceed 32)	maximum number of program matrices	2.14.6	-
MAX_PROGRAM_MATRIX_STACK_DEPTH_ARB	Z+	GetIntegerv	1	maximum program matrix stack depth	2.14.6	-
MAX_PROGRAM_INSTRUCTIONS_ARB	Z+	GetProgramivARB	128	maximum program instructions	6.1.12	-
MAX_PROGRAM_TEMPORARIES_ARB	Z+	GetProgramivARB	12	maximum program temporaries	6.1.12	-
MAX_PROGRAM_PARAMETERS_ARB	Z+	GetProgramivARB	96	maximum program parameter bindings	6.1.12	-
MAX_PROGRAM_ATTRIBS_ARB	Z+	GetProgramivARB	16	maximum program attribute bindings	6.1.12	-
MAX_PROGRAM_ADDRESS_REGISTERS_ARB	Z+	GetProgramivARB	1	maximum program address registers	6.1.12	-
MAX_PROGRAM_NATIVE_INSTRUCTIONS_ARB	Z+	GetProgramivARB	-	maximum program native instructions	6.1.12	-
MAX_PROGRAM_NATIVE_TEMPORARIES_ARB	Z+	GetProgramivARB	-	maximum program native temporaries	6.1.12	-
MAX_PROGRAM_NATIVE_PARAMETERS_ARB	Z+	GetProgramivARB	-	maximum program native parameter bindings	6.1.12	-
MAX_PROGRAM_NATIVE_ATTRIBS_ARB	Z+	GetProgramivARB	-	maximum program native attribute bindings	6.1.12	-
MAX_PROGRAM_NATIVE_ADDRESS_REGISTERS_ARB	Z+	GetProgramivARB	-	maximum program native address registers	6.1.12	-

Table X.11. New Implementation-Dependent Values Introduced by ARB_vertex_program. Values queried by GetProgramivARB require a <pname> of VERTEX_PROGRAM_ARB.

Revision History

Rev.	Date	Author	Changes
46	07/25/07	mjk	Document how the ARB and NV generic arrays interact. This documents NVIDIA's long-standing implemented behavior.
45	09/27/04	pbrown	Fixed GLX protocol, removing the unused <pname> parameters for GetProgram{Env,Local}Parameter [df]vARB, leaving an unused CARD32 in its place. This was an error when propogating NV_vertex_program protocol, which did have a <pname> parameter.
44	09/12/03	pbrown	Fixed opcode table entry for "ARL" -- it takes a scalar operand as specified in the grammar.
43	08/17/03	pbrown	Fixed a couple minor typos (missing quotes) in the grammar.
42	05/01/03	pbrown	Clarified the handling of color sum; old text suggested that COLOR_SUM controlled the operation even when doing separate specular lighting.

```
    41    04/18/03  pbrown    Add a couple overlooked contributors.

    40    03/03/03  pbrown    Fixed list of immediate-mode VertexAttrib
                              functions in Section 2.7 -- there are no
                              normalized float functions (e.g.,
                              VertexAttrib4Nfv).  Clarified issue (42)
                              describing how point size is handled in vertex
                              program mode.

    39    01/31/03  pbrown    Fixed minor bug in the description of vertex
                              array state kept by the GL -- normalization
                              flags were omitted from the text (but were in
                              the state tables).

    38    01/08/03  pbrown    Fixed bug where "state.matrix.mvp" was specified
                              incorrectly -- it should be P*M0 rather than
                              M0*P.

    37    12/23/02  pbrown    Fixed minor typos.  Fixed state table bug where
                              CURRENT_VERTEX_ATTRIB_ARB was incorrectly called
                              CURRENT_ATTRIB_ARB.

    36    09/09/02  pbrown    Fixed incorrect example of matrix row bindings
                              (and transposition).  Small wording/typo fixes.

    35    08/27/02  pbrown    Fixed several minor typos.  Documented that a
                              program string should not include a null
                              terminator in its first <len> characters.  Fixed
                              dangling reference in <paramMultipleItem>
                              grammar rule.  Fix incorrect wording in
                              computation of state.light.half vector.
                              Documented that the inverse of a singular matrix
                              is undefined.  Clarified that native
                              instructions can include additions due to
                              emulation of features not supported natively.
                              Documented that LG2 produces undefined results
                              with zero or negative inputs.  Clarified that
                              POW may be a LOG/MUL/EXP sequence, but isn't
                              necessarily so.  Disallowed multiple modelview
                              matrix syntax if ARB_vertex_blend or
                              EXT_vertex_weighting is unsupported.  Fixed
                              state table query function for attribute array
                              enables.  Added missing state table entry for
                              PROGRAM_UNDER_NATIVE_LIMITS_ARB.

    34    07/19/02  pbrown    Fixed typo in ArrayElement pseudo-code.

    33    07/17/02  pbrown    Fixed bug in the <stateLModProperty> grammar
                              rule.  Fixed documentation to indicate that
                              Enable/DisableVertexAttribArray are not display
                              listable.

    31    07/15/02  pbrown    Fixed <SWZ_instruction> grammar rule to match
                              the spec language -- base operand negation
                              doesn't apply, since you can independently
                              negate components.  Modified "XPD" instruction
```

```
                          to eliminate the implicit masking of the "w"
                          component; slight efficiency gain for some SW
                          implementations.  Modified "scenecolor" binding
                          to pass the diffuse alpha instead of the ambient
                          alpha; the former is more useful.

    30    07/02/02  pbrown    Minor wording fixes.

    29    06/21/02  pbrown    Mostly minor bug fixes from reviewer feedback;
                              also added one new item approved at ARB meeting.

                              Additions:  Added a "lightmodel.*.scenecolor"
                              holding lit color containing the composite
                              lighting result ignoring individual lights --
                              i.e., from only emissive materials and the light
                              model.  Added GLX protocol.

                              Minor changes:  Numerous minor typo and wording
                              fixes.  Added missing vertex array types to
                              vertex array size/type/normalized table.  Added
                              missing description of ambient light model color
                              binding.  Removed several references to language
                              features long since deleted.  Documented that
                              POW is not necessarily implemented as
                              LOG/MUL/EXP.  Fixed a couple minor errata in the
                              EXT_vertex_shader interaction section.  Added a
                              list of reserved keywords.

    28    06/16/02  pbrown    Minor updates based on feedback given on
                              versions 26 and 27.

                              Additions:  Added section on EXT_vertex_shader
                              interaction, provided by ATI.

                              Minor changes:  Minor grammar and readability
                              fixes.  Fixed several incomplete definitions.
                              Removed "GL" and "gl" prefixes from several
                              enumerants and function names to match spec
                              conventions.  Clarified the precision issue on
                              EX2/LG2.  Added missing functions that take
                              VERTEX_PROGRAM_ARB.  Clarified component
                              normalization on vertex arrays.  Clarified
                              clipping section to note that user clipping is
                              done with position invariant programs.
                              Clarified the handling of program zero in
                              BindProgramARB.  Fixed a couple incorrect
                              grammar rules.  Fixed incorrect grammar
                              references in description of vertex program
                              parameter array accesses.  Documented that the
                              SWZ instruction doesn't take "normal" swizzle
                              and negation modifiers, since it already has
                              some.  Clarified some NV_vertex_program
                              interactions.

    27    06/07/02  pbrown    Minor update based on ARB_vertex_program sample
                              implementation work.
```

```
                            Changes:  Changed fog coordinate attribute and
                            result binding name to "fogcoord" (was "fog").
                            Rearranged grammar based on sample
                            implementation verification.  There might be a
                            minor fix or two stuck in there.

      26    06/04/02  pbrown    Spec checkpoint published on the working group
                            web site.  Resolves most of the remaining open
                            issues.

                            Deletions:  Removed the ability to bind the
                            color matrix (from ARB_imaging).

                            Changes:  Resolved the handling of vertex
                            attribute zero (it always specifies a vertex, in
                            program mode or not).  Resolved the handling of
                            generic and conventional vertex attribute arrays
                            (they are always sent, although they also have
                            "undefined aliasing" behavior).  Default values
                            of generic attributes are undefined, to
                            accommodate aliasing and non-aliasing
                            implementations.  Added pseudocode to document
                            the processing of ArrayElement.  Moved program
                            object language into the vertex program section.
                            Renamed the fog coordinate attribute and result
                            binding to "fogcoord".  Added missing
                            documentation of the agreed-upon semantic
                            restriction that programs can't bind
                            conventional / generic attribute pairs that may
                            alias.  Added documentation of what happens when
                            multiple contexts share program objects
                            Disallowed queries of generic attribute zero.

                            Fixes:  Fixed prototype for VertexAttrib4Nub.

                            Minor Changes:  Minor typo and language
                            fixes. Added guidelines for future
                            programmability extensions.  Added several
                            missing grammar rules.

      25    05/30/02  pbrown    Spec checkpoint published on the working group
                            web site.

                            Additions:  Add "DPH" (dot product homogeneous)
                            instruction.  Added the ability to query the
                            current matrix in transposed form.  Assigned
                            enumerant values for program matrices.  Added
                            the ability bind selected rows of a matrix.
                            Added ability to bind matrix palette matrices.

                            Changes:  Renamed PROGRAM_NAME_ARB to
                            PROGRAM_BINDING_ARB.  Specifying the number of
                            elements in parameter arrays is now optional,
                            but compilation will fail if the specified count
                            does not match.  Programs performing
                            out-of-bounds array accesses using absolute
                            addressing will now fail to load.  Allow "$" in
```

```
                                 token names.

                                 Minor changes:  Completed scrub of the issues
                                 and error list.  Added new issues about reserved
                                 keywords, identifier characters, and parsing of
                                 floating-point constants in programs.
                                 Miscellaneous typo fixes.  Updated the grammar
                                 to include light products and half angles, moved
                                 material properties from per-vertex to parameter
                                 bindings, and a few other miscellaneous fixes.
                                 Simplified the matrix binding table.  Modified
                                 the color sum portion of the spec to explicitly
                                 add R,G,B only.  Removed several incorrect
                                 errors.  Fixed program object state table.

    24    05/21/02  pbrown    Spec checkpoint published on the working group
                                 web site.

                                 Deletions:  Removed the semantic requirement
                                 that vertex programs write a vertex position,
                                 per working group resolution.

                                 Minor changes:  Cleaned up cruft in a number of
                                 issues; many more to go.  Added several issues.
                                 Documented that VertexAttrib functions are
                                 allowed inside Begin/End pairs.  Changed default
                                 initialization values of generic attributes to
                                 accommodate attribute aliasing.  Documented that
                                 point sizes and fog coordinates computed by
                                 vertex programs are clipped during primitive
                                 clipping.  Documented that vertex program
                                 behavior is undefined in color index mode.

    23    05/21/02  pbrown    Spec checkpoint.  More changes from working
                                 group deliberations.

                                 Additions:  Added vertex materials as allowed
                                 program parameter bindings.  Allow programs to
                                 use vertex attribute binding names, program
                                 parameter binding names, result variable binding
                                 names, and constants in executable statements,
                                 resulting in implicit bindings.  Added support
                                 for binding a single row of a matrix.  Added
                                 support for binding precomputed light/material
                                 products.  Added restriction that a single GL
                                 state vector can't be bound multiple times in
                                 two separate arrays accessed with relative
                                 addressing.  Added new section documenting the
                                 various resource limits, and introducting the
                                 idea of "native" resource limits and counts.

                                 Deletions:  Removed vertex materials as allowed
                                 vertex attribute bindings.

                                 Minor changes:  Added more names to the
                                 contributors list.  Updated issues concerning
                                 undefined aliasing.  Moved NV_vertex_program
```

```
                          related issues to the NV_vertex_program
                          interaction sections.  Updated NV_vertex_program
                          interactions.  Updated lighting example using
                          new derived state bindings.  Clarified that
                          "!!ARBvp1.0" is not a token in the grammar and
                          that programs are parsed beginning immediately
                          after the header string.  Added text to explain
                          all attribute, program parameter, and result
                          bindings instead of depending on binding table
                          interpretations.  Broke the large program
                          parameter binding table into several smaller
                          tables, organized by function.  Documented that
                          the queryable program error string may contain
                          warnings when a program loads successfully, and
                          that a queried program error string is
                          guaranteed to remain constant only until the
                          next program load attempt.  Added PROGRAM_NAME
                          query to the appropriate state table.

      22    05/20/02  pbrown    Spec checkpoint.  More changes from working
                          group deliberations.

                          Added functionality:  Assigned enumerant values.
                          Added "undefined (vertex attribute) aliasing"
                          language, where setting a generic attribute
                          leaves a conventional one undefined, and vice
                          versa.  Added support for matrix indices from
                          ARB_matrix_palette.  Added default program
                          object zero.  Added support for simple named
                          variable aliasing.  Added queries of API-level
                          and "native" resources used by a program and
                          their corresponding limits.  Added general query
                          to determine if a program fits in native limits.

                          Removed functionality:  Removed extension string
                          entry for position-invariant programs (now
                          mandatory).

                          Modified functionality:  GetProgram and
                          GetProgramString now take a target instead of a
                          program name.  Default values for 3 generic
                          attributes are changed for consistent aliasing.
                          Added 1/(end-start) binding for fog parameters.
                          Added precomputed infinite light/viewer half
                          angle binding.  ProgramString takes a "void *"
                          instead of a "ubyte *".

                          Minor Changes:  Clarified key terms for the
                          extension.  Documented that user clipping is not
                          supported in the base extension.  Added warnings
                          on a couple pitfalls from uninitalized result
                          registers.  Document that EXT_vertex_weighting
                          and ARB_vertex_blend use the same weight.
                          Cleaned up bindings for 4-component colors for
                          cases where only three components are used.
                          Documented the implicit absolute value operation
                          on the LOG instruction.  Renamed query token for
```

```
                                querying generic vertex attribute array enables.
                                Renamed and relocated vertex program binding
                                query.  Added language to Section 2.6.  Changed
                                syntax to bind a range of the environment or
                                local paramater array to use double dots ("..").
                                Clarified what happens on a weight binding using
                                more weights than an implementation supports.
                                Clarified the component selection pseudocode for
                                scalar operand loads.  Clarified what happens to
                                vertex program results during primitive
                                assembly.  Fixed a number of errors in the state
                                tables.

    21    04/29/02  pbrown    More changes from working group deliberations.

                                Added functionality:  Added "FLR", "FRC", "POW",
                                and "XPD" (cross product) instructions.  Added
                                functions to enable/disable generic attribute
                                arrays.  Added query of a program error string.
                                Added "format" enum argument to ProgramStringARB
                                to provide for possible programs not using ASCII
                                text.  Added new enums to permit different
                                limits for overall numbers of program
                                environment and local parameters and the number
                                of parameters that can be bound by a program.

                                Removed functionality:  Removed support for
                                evaluators for generic attributes.  Removed
                                support for program residency management.
                                Removed support for user clipping in standard
                                vertex programs.  Removed functionality to set
                                more than one program environment parameter at
                                once.

                                Issues/Changes:  Resolved set of immediate mode
                                VertexAttrib functions.  Combined parameter
                                bindings for several groups of related GL state.
                                Resolved user clipping issue by disallowing
                                except for position invariant programs.
                                Resolved limits for array relative offsets.
                                GenProgramsARB and DeleteProgramsARB will use
                                texture object model.  Program environment
                                parameters will not be pushed/popped.

                                Bug fixes:  Fixed vertex attribute index
                                prototypes (should be uint instead of int).
                                Fixed tokens used to query generic attribute
                                state (should have VERTEX prefixes).  Fixed
                                documentation of the alpha component of material
                                colors.  Fixed documentation of initial state
                                for vertex program objects.

                                Temporarily removed dated GLX protocol language
                                (will restore in one pass after resolving
                                remaining issues).

    20    04/17/02  pbrown    Clarify the meaning of individual components of
```

```
                            program parameters where the component mapping
                            is not obvious from the mapping table.

  18    04/15/02  pbrown    Update spec to reflect issues resolved by the
                            working group on 4/11.

                            Started using "program matrix" terminology --
                            was "tracking matrix".

                            Address register variables must now be declared.
                            The number of address registers can be queried.
                            Only 1-component address registers are currently
                            supported.

                            VertexAttribPointer takes a separate argument to
                            indicate normalized data, now called
                            "normalized" (was "normalize").

                            ProgramString and functions to set and query
                            local parameters all take a <target> and refer
                            to the currently bound program (previously took
                            a program number).  Have not touched other
                            somewhat related issues (e.g., is there a
                            program object zero?).

                            Added COLOR_SUM enable (taken directly from
                            EXT_secondary_color) for completeness and a
                            few updates to EXT_secondary_color
                            interactions.

                            Fixed cut-and-paste error in specification of
                            the clip-space user clip dot product.

                            Documented special-case arithmetic for ADD, MAD,
                            and MUL.

                            Eliminated some wordiness in DP3 and DP4
                            instruction pseudo-code.

                            Minor changes not from working group:  More
                            verbose documentation on the user clipping
                            issue.  More detail on other opcode candidates.
                            Removed redundant color material issue.  Minor
                            fixes to error roundup (not complete) and to
                            state tables to reflect that most program
                            execution variables are initially undefined.

  17    04/08/02  pbrown    Issues:  Enumerated other candidates for
                            consideration in the instruction set -- there
                            may be more that I missed.  Added a description
                            of some of the considerations on how color
                            material should be treated.  Added issues on the
                            name of the program matrices, the number of MVP
                            matrices, and where variable declarations
                            can be done.  Added numbers to all spec issues.
                            Fixed lighting example (issue 74) so it
                            compiles, and so that the half vector is
```

```
                                properly normalized.

                                Grammar:  Eliminated stale hardwired temporary,
                                parameter array, and result register names
                                (R<n>, c[<n>], and o[...]).  Should have been
                                deleted going from revision 5 to revision 12.
                                Added missing program matrix bindings to the
                                grammar.  Eliminated state material-as-parameter
                                bindings.  Fixed texgen paramete bindings, which
                                should have had both "eye" and "object" planes.
                                Added separate address register write masks and
                                selectors to reflect the current single-
                                component address register restriction.  Added
                                an array[A0.x] rule -- before, you erroneously
                                had to add or subtract a constant.  Modified SWZ
                                so that the register being swizzled can't take a
                                conventional swizzle suffix, too.

                                Also reorganized grammar to closely mirror the
                                sample implementation, consolidating a number of
                                redundant rules.  Also fixed several bugs
                                found by the implementation.

                                Documentation changes to "LIT" to use the right
                                variable name and also indicate that 0^0=1.
                                Fixed the computation of result.y in the "LOG"
                                instruction.

                                Other:  Added dependency on ARB_imaging.  Added
                                notation of Microsoft's IP claims.  Fixed name
                                of MAX_VERTEX_PROGRAM_TEMPORARIES_ARB.

                                A few minor typo fixes.

    12    03/11/02  pbrown    Modified spec to reflect decisions made at
                                the March 2002 ARB meeting.  Distributed
                                to the OpenGL participants list.

    5     03/03/02  pbrown    Distributed to the ARB prior to March 2002
                                ARB meeting.
```

Name

    ARB_window_pos

Name Strings

    GL_ARB_window_pos

Status

    Complete. Approved by ARB on February 14, 2002.

Version

    Last Modified Date: June 11, 2002

Number

    ARB Extension #25

Dependencies

    OpenGL 1.0 is required.
    The extension is written against the OpenGL 1.3 Specification
    GL_EXT_fog_coordinate effects the definition of this extension.
    GL_EXT_secondary_color effects the definition of this extension.

Overview

In order to set the current raster position to a specific window coordinate with the RasterPos command, the modelview matrix, projection matrix and viewport must be set very carefully. Furthermore, if the desired window coordinate is outside of the window's bounds one must rely on a subtle side-effect of the Bitmap command in order to avoid frustum clipping.

This extension provides a set of functions to directly set the current raster position in window coordinates, bypassing the modelview matrix, the projection matrix and the viewport-to-window mapping. Furthermore, clip testing is not performed, so that the current raster position is always valid.

This greatly simplifies the process of setting the current raster position to a specific window coordinate prior to calling DrawPixels, CopyPixels or Bitmap. Many matrix operations can be avoided when mixing 2D and 3D rendering.

IP Status

No IP issues.

Issues

(1) Should we offer all 24 entrypoints, just like glRasterPos?

RESOLVED. No. Don't implement the 4-coordinate functions as they're really useless. However, we will implement the short and double-type functions for completeness.

For example, it's conceivable that an application may have data structures encoding window coordinates as a 2- or 3-vector of shorts and will want to use WindowPos3svARB(). Chris Hecker lobbied for this on the grounds of orthogonality.

(2) Should we have unique GLX protocol requests for every entrypoint or just a 3-float version?

RESOLVED. Just a 3-float version will suffice since all reasonable window coordinate values can be perfectly represented with single-precision floating point.

*(4) For WindowPos2*ARB(), is zero the correct value for z? Afterall, z is a window coordinate, not an object coordinate.*

RESOLVED. Yes, zero is correct. Zero corresponds to the front of the depth range. That's where one would usually want Bitmap, DrawPixels and CopyPixels to be positioned in z when rendering 2D primitives over a 3D scene.

(5) What about glDepthRange?

RESOLVED. Map the WindowPos z value into the range specified by DepthRange. There's a popular optimization used to avoid depth buffer clears for scenes that completely fill the window in which the depth buffer is effectively halfed and reversed in alternate frames by calling DepthRange. The WindowPos z value should be subjected to depth range mapping so that it will work with this optimization, and in other scenarios.

(6) Should we mention EXT_fog_coord and EXT_secondary_color in this extension?

RESOLVED. Yes, otherwise implementors may not know what to do with them. It's been suggested that we instead go back and update the EXT_fog_coordinate and EXT_secondary_color specifications with respect to ARB_window_pos instead. However, that seems unlikely to happen and seems error-prone/obscure for implementors.

(7) What about the raster fog coordinate?

RESOLVED. If EXT_fog_coord is not supported, CURRENT_RASTER_DISTANCE is set to zero.

If EXT_fog_coord is supported, the behavior is dependent on the current state of FOG_COORDINATE_SOURCE_EXT. If the fog coordinate source is FRAGMENT_DEPTH_EXT, CURRENT_RASTER_DISTANCE is set to zero. If the fog coordinate source is FOG_COORDINATE_EXT, CURRENT_RASTER_DISTANCE is set to the current fog coordinate.

The value chosen for CURRENT_RASTER_DISTANCE state matches the value that would be chosen for normal vertices, except that WindowPos does not allow the GL to compute eye coordinates that would be used to generate a fog distance value. Instead, a value of zero is always used as a fog distance.

With the current EXT_fog_coord specification, there are two pieces of RasterPos state that drive fog (CURRENT_RASTER_DISTANCE and the current raster fog coordinate). The setting of the fog coordinate source selects which piece of state is used at rasterization (Bitmap, DrawPixels) time. Instead, this extension moves the selection of fog state to RasterPos state computation instead of rasterization and combines the two pieces of state into a single CURRENT_RASTER_DISTANCE.

Current implementations of EXT_fog_coord that support two pieces of state can either change the implementations to merge the two pieces into a single state or contiue to maintain two pieces of state. If the implementations continue to maintain two pieces of state, both the CURRENT_RASTER_DISTANCE and current raster fog coordinate are set to the same value.

(8) What about the secondary raster color?

RESOLVED. If EXT_secondary_color is supported, the (unnamed) current raster secondary color is set by taking the current secondary color and clamping the components to the range [0,1].

If EXT_secondary_color is not supported, the current raster secondary color is set to (0,0,0).

(9) How is this extension specification different from the MESA_window_pos extension?

(a) Clarified that lighting and texgen aren't used when updating the current raster state.

(b) Explicitly state the effect on CURRENT_RASTER_DISTANCE and CURRENT_RASTER_POSITION_VALID.

(c) Explain how the raster position's secondary color and fog coordinate are handled.

(d) Z is mapped according to the DEPTH_RANGE values.

(e) Removed the functions which take 4 coordinates.

New Procedures and Functions

```
    void WindowPos2dARB(double x, double y)
    void WindowPos2fARB(float x, float y)
    void WindowPos2iARB(int x, int y)
    void WindowPos2sARB(short x, short y)
    void WindowPos2dvARB(const double *p)
    void WindowPos2fvARB(const float *p)
    void WindowPos2ivARB(const int *p)
    void WindowPos2svARB(const short *p)

    void WindowPos3dARB(double x, double y, double z)
    void WindowPos3fARB(float x, float y, float z)
    void WindowPos3iARB(int x, int y, int z)
    void WindowPos3sARB(short x, short y, short z)

    void WindowPos3dvARB(const double *p)
    void WindowPos3fvARB(const float *p)
    void WindowPos3ivARB(const int *p)
    void WindowPos3svARB(const short *p)
```

New Tokens

none

Additions to Chapter 2 of the OpenGL 1.3 Specification (OpenGL Operation)

In **section 2.12 (Current Raster Position)**, p. 42, insert after fifth paragraph:

Alternately, the current raster position may be set by one of the WindowPosARB commands:

```
   void WindowPos{23}{ifds}ARB( T coords );
   void WindowPos{23}{ifds}vARB( const T coords );
```

WindosPos3ARB takes three values indicating x, y and z while WindowPos2ARB takes two values indicating x and y with z implicitly set to 0.

The CURRENT_RASTER_POSITION, (RPx, RPy, RPz, RPw), is updated as follows:

```
    RPx = x

    RPy = y

          { n,                  if z <= 0
    RPz = { f,                  if z >= 1
          { n + z * (f - n),   otherwise

    RPw = 1
```

where <n> is the DEPTH_RANGE's near value, and <f> is the DEPTH_RANGE's far value.

In RGBA mode, CURRENT_RASTER_COLOR is updated from CURRENT_COLOR with each color component clamped to the range [0,1].

In color index mode, CURRENT_RASTER_INDEX is updated from CURRENT_INDEX.

```
        All sets of CURRENT_RASTER_TEXTURE_COORDS are updated from
        the corresponding CURRENT_TEXTURE_COORDS sets.

        CURRENT_RASTER_POSITION_VALID is set to TRUE.

        If EXT_fog_coord is not supported.

          CURRENT_RASTER_DISTANCE is set to zero.

        If EXT_fog_coord is supported:

          CURRENT_RASTER_DISTANCE is set to

            { CURRENT_FOG_COORDINATE, if FOG_COORDINATE_SOURCE_EXT is set
            {                           to FOG_COORDINATE_EXT, or
            { 0,                      if FOG_COORDINATE_SOURCE_EXT is set
            {                           to FRAGMENT_DEPTH_EXT.

        If EXT_secondary_color is supported:

          The current raster secondary color is set by clamping the components
          of CURRENT_SECONDARY_COLOR_EXT to [0,1], if in RGBA mode.

        If EXT_secondary_color is not supported:

         The current raster secondary color (the secondary color used for all
         pixel and bitmap rasterization) is set to (0,0,0), if in RGBA mode.

        Note that lighting, texture coordinate generation, and clipping are
        not performed by the WindowPos*ARB functions.
```

Additions to Chapter 5 of the OpenGL 1.3 Specification (Special Functions)

In **section 5.2 (Selection)**, p. 188, modify the fourth paragraph to read:

```
        In selection mode, if a point, line, polygon, or the valid
        coordinates produced by a RasterPos command intersects the clip
        volume (section 2.11) then this primitive (or RasterPos command)
        causes a selection hit.  WindowPos commands always generate a
        selection hit since the resulting raster position is always
        valid.  In the case of polygons (...)
```

Additions to the AGL/GLX/WGL Specifications

None

GLX Protocol

One new GL rendering command is added. The following command is sent to the server as part of a glXRender request:

```
        WindowPosARB
            2           16              rendering command length
            2           230             rendering command opcode
            4           FLOAT32         x
            4           FLOAT32         y
            4           FLOAT32         z
```

Errors

    INVALID_OPERATION is generated if WindowPosARB is called betweeen
    Begin and End.

New State

    None.

New Implementation Dependent State

    None.

Revision History

```
    May 17, 2001
        - Initial version based on GL_MESA_window_pos extension
    May 22, 2001
        - Explicitly state that x, y, z are window coordinates and w is
          a clip space coordinate.  (Dan Brokenshire)
    May 23, 2001
        - Resolved issues 1 and 2.
        - Added issues 4 and 5.
    May 24, 2001
        - Rewrote body of specification to more clearly indicate how all
          raster position state is updated by WindowPos.
        - Updated the issues section.
    Jun 13, 2001
        - Added back the double and short versions of WindowPos()
        - Added fog coord issue and discusstion.
        - Reordered/renumbered the issues section.
    Jun 22, 2001
        - Set raster secondary color to current secondary color, not black
    Jun 25, 2001
        - Another change to secondary color, think I got it now!
    Nov 16, 2001
        - updated email address
        - List options "A" and "B" to determine behaviour of current raster
          fog coordinate.
    Nov 17, 2001
        - minor clean-ups
    Dec 12, 2001
        - rewrite against the OpenGL 1.3 spec
        - fixed a few typos
    Jan 10, 2002
        - update the interaction with EXT_fog_coord and EXT_secondary_color
          based on the proposed resolution from the December 2001 ARB
          meeting. (Pat Brown)
    Jan 18, 2002
        - Merges two pieces of fog state into a single state. (Bimal Poddar)
    Mar 12, 2002
        - Added GLX protocol. (Jon Leech)
    June 11, 2002
        - Clarifications: RGBA/index color updates apply only in
          RGBA/index mode respectively. Hits are generated in selection mode.
```

Name

    ATI_draw_buffers

Name Strings

    GL_ATI_draw_buffers

Status

    Complete.

Version

    Last Modified Date: December 30, 2002
    Revision: 8

Number

    277

Dependencies

    The extension is written against the OpenGL 1.3 Specification.

    OpenGL 1.3 is required.

    ARB_fragment_program affects the definition of this extension.

Overview

    This extension extends ARB_fragment_program to allow multiple output
    colors, and provides a mechanism for directing those outputs to
    multiple color buffers.

Issues

    (1) How many GL_DRAW_BUFFER#_ATI enums should be reserved?

      RESOLVED: We only need 4 currently, but for future expandability
      it would be nice to keep the enums in sequence. We'll specify
      16 for now, which will be more than enough for a long time.

New Procedures and Functions

    void DrawBuffersATI(sizei n, const enum *bufs);

New Tokens

    Accepted by the <pname> parameters of GetIntegerv, GetFloatv,
    and GetDoublev:

MAX_DRAW_BUFFERS_ATI	0x8824
DRAW_BUFFER0_ATI	0x8825
DRAW_BUFFER1_ATI	0x8826
DRAW_BUFFER2_ATI	0x8827
DRAW_BUFFER3_ATI	0x8828
DRAW_BUFFER4_ATI	0x8829
DRAW_BUFFER5_ATI	0x882A
DRAW_BUFFER6_ATI	0x882B
DRAW_BUFFER7_ATI	0x882C
DRAW_BUFFER8_ATI	0x882D
DRAW_BUFFER9_ATI	0x882E
DRAW_BUFFER10_ATI	0x882F
DRAW_BUFFER11_ATI	0x8830
DRAW_BUFFER12_ATI	0x8831
DRAW_BUFFER13_ATI	0x8832
DRAW_BUFFER14_ATI	0x8833
DRAW_BUFFER15_ATI	0x8834

Additions to Chapter 2 of the OpenGL 1.3 Specification (OpenGL Operation)

    None

Additions to Chapter 3 of the OpenGL 1.3 Specification (Rasterization)

Modify Section 3.11.2, Fragment Program Grammar and Semantic Restrictions

    (replace <resultBinding> grammar rule with these rules)

```
<resultBinding>        ::= "result" "." "color" <optOutputColorNum>
                         | "result" "." "depth"

<optOutputColorNum>    ::= ""
                         | "[" <outputColorNum> "]"

<outputColorNum>       ::= <integer> from 0 to MAX_DRAW_BUFFERS_ATI-1
```

Modify Section 3.11.3.4, Fragment Program Results

    (modify Table X.3)

Binding	Components	Description
result.color[n]	(r,g,b,a)	color n
result.depth	(*,*,*,d)	depth coordinate

Table X.3: Fragment Result Variable Bindings. Components labeled "*" are unused. "[n]" is optional -- color <n> is used if specified; color 0 is used otherwise.

(modify third paragraph) If a result variable binding matches "result.color[n]", updates to the "x", "y", "z", and "w" components of the result variable modify the "r", "g", "b", and "a" components, respectively, of the fragment's corresponding output color. If "result.color[n]" is not both bound by the fragment program and written by some instruction of the program, the output color <n> of the fragment program is undefined.

Add a new Section 3.11.4.5.3

3.11.4.5.3 Draw Buffers Program Option

If a fragment program specifies the "ATI_draw_buffers" option, it will generate multiple output colors, and the result binding "result.color[n]" is allowed, as described in section 3.11.3.4, and with modified grammar rules as set forth in section 3.11.2. If this option is not specified, a fragment program that attempts to bind "result.color[n]" will fail to load, and only "result.color" will be allowed.

Additions to Chapter 4 of the OpenGL 1.3 Specification (Per-Fragment Operations and the Frame Buffer)

Modify Section 4.2.1, Selecting a Buffer for Writing (p. 168)

(modify the title and first paragraph, p. 168)

4.2.1 Selecting Color Buffers for Writing

The first such operation is controlling the color buffers into which each of the output colors are written. This is accomplished with either DrawBuffer or DrawBuffersATI. DrawBuffer defines the set of color buffers to which output color 0 is written.

(insert paragraph between first and second paragraph, p. 168)

DrawBuffer will set the draw buffer for output colors other than 0 to NONE. DrawBuffersATI defines the draw buffers to which all output colors are written.

```
  void DrawBuffersATI(sizei n, const enum *bufs);
```

<n> specifies the number of buffers in <bufs>. <bufs> is a pointer to an array of symbolic constants specifying the buffer to which each output color is written. The constants may be NONE, FRONT_LEFT, FRONT_RIGHT, BACK_LEFT, BACK_RIGHT, and AUX0 through AUXn, where n + 1 is the number of available auxiliary buffers. The draw buffers being defined correspond in order to the respective output colors. The draw buffer for output colors beyond <n> is set to NONE.

If the "ATI_draw_buffers" fragment program option, is not being used then DrawBuffersATI specifies a set of draw buffers into which output color 0 is written.

The maximum number of draw buffers is implementation dependent and must be at least 1. The number of draw buffers supported can be queried with the state MAX_DRAW_BUFFERS_ATI.

The constants FRONT, BACK, LEFT, RIGHT, and FRONT_AND_BACK that refer to multiple buffers are not valid for use in DrawBuffersATI and will result in the error INVALID_OPERATION.

If DrawBuffersATI is supplied with a constant (other than NONE) that does not indicate any of the color buffers allocated to the GL context, the error INVALID_OPERATION will be generated. If <n> is greater than MAX_DRAW_BUFFERS_ATI, the error INVALID_OPERATION will be generated.

(replace last paragraph, p. 169)

The state required to handle color buffer selection is an integer for each supported output color. In the initial state, draw buffer for output color 0 is FRONT if there are no back buffers; otherwise it is BACK. The initial state of draw buffers for output colors other then 0 is NONE.

Additions to Chapter 5 of the OpenGL 1.3 Specification (Special Functions)

None

Additions to Chapter 6 of the OpenGL 1.3 Specification (State and State Requests)

None

Dependencies on ARB_fragment_program

If ARB_fragment_program is not supported then all changes to section 3.11 are removed.

Interactions with possible future extensions

If there is some other future extension that defines multiple color outputs then this extension and glDrawBuffersATI could be used to define the destinations for those outputs. This extension need not be used only with ARB_fragment_program.

Errors

The error INVALID_OPERATION is generated by DrawBuffersATI if a color buffer not currently allocated to the GL context is specified.

The error INVALID_OPERATION is generated by DrawBuffersATI if <n> is greater than the state MAX_DRAW_BUFFERS_ATI.

The error INVALID_OPERATION is generated by DrawBuffersATI if value in <bufs> does not correspond to one of the allowed buffers.

New State

    (table 6.19, p227) add the following entry:

Get Value	Type	Get Command	Initial Value	Description	Section	Attribute
DRAW_BUFFERi_ATI	Z10*	GetIntegerv	see 4.2.1	Draw buffer selected for output color i	4.2.1	color-buffer

New Implementation Dependent State

Get Value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
MAX_DRAW_BUFFERS_ATI	Z+	GetIntegerv	1	Maximum number of active draw buffers	4.2.1	-

Revision History

```
   Date: 12/30/2002
   Revision: 8
      - Clarified that DrawBuffersATI will set the set of draw buffers to
        write color output 0 to when the "ATI_draw_buffer" fragments
        program option is not in use.

   Date: 9/27/2002
   Revision: 7
      - Fixed confusion between meaning of color buffer and draw buffer
        in last revision.
      - Fixed mistake in when an error is generated based on the <n>
        argument of DrawBuffersATI.

   Date: 9/26/2002
   Revision: 6
      - Cleaned up and put in sync with latest ARB_fragment_program
        revision (#22).  Some meaningless changes made just in the name
        of consistency.

   Date: 9/11/2002
   Revision: 5
      - Added section 3.11.4.5.3.
      - Added enum numbers to New Tokens.

   Date: 9/9/2002
   Revision: 4
      - Changed error from MAX_OUTPUT_COLORS to MAX_DRAW_BUFFERS_ATI.
      - Changed 3.10 section numbers to 3.11 to match change to
        ARB_fragment_program spec.
      - Changed ARB_fragment_program from required to affects, and
        added section on interactions with it and future extensions
        that define multiple color outputs.

   Date: 9/6/2002
   Revision: 3
      - Changed error to INVALID OPERATION.
      - Cleaned up typos.
```

```
    Date: 8/19/2002
    Revision: 2
       - Added a paragraph that specifically points out that the
         constants that refer to multiple buffers are not allowed with
         DrawBuffersATI.
       - Changed bufs to <bufs> in a couple of places.

    Date: 8/16/2002
    Revision: 1
       - First draft for circulation.
```

Name

    ATI_texture_float

Name Strings

    GL_ATI_texture_float

Status

    Complete.

Version

    Last Modified Date: December 4, 2002
    Revision: 4

Number

    280

Dependencies

    OpenGL 1.1 or EXT_texture is required.

    The extension is written against the OpenGL 1.3 Specification.

Overview

    This extension adds texture internal formats with 32 and 16 bit
    floating-point components. The 32 bit floating-point components
    are in the standard IEEE float format. The 16 bit floating-point
    components have 1 sign bit, 5 exponent bits, and 10 mantissa bits.
    Floating-point components are clamped to the limits of the range
    representable by their format.

Issues

    1. *Should we expose a GL_FLOAT16_ATI pixel type so that the 16 bit*
       float textures can be directly loaded?

       RESOLUTION: This will be exposed in a separate extension.

New Procedures and Functions

   None

New Tokens

Accepted by the <internalFormat> parameter of TexImage1D, TexImage2D, and TexImage3D:

RGBA_FLOAT32_ATI	0x8814
RGB_FLOAT32_ATI	0x8815
ALPHA_FLOAT32_ATI	0x8816
INTENSITY_FLOAT32_ATI	0x8817
LUMINANCE_FLOAT32_ATI	0x8818
LUMINANCE_ALPHA_FLOAT32_ATI	0x8819
RGBA_FLOAT16_ATI	0x881A
RGB_FLOAT16_ATI	0x881B
ALPHA_FLOAT16_ATI	0x881C
INTENSITY_FLOAT16_ATI	0x881D
LUMINANCE_FLOAT16_ATI	0x881E
LUMINANCE_ALPHA_FLOAT16_ATI	0x881F

Additions to Chapter 2 of the OpenGL 1.3 Specification (OpenGL Operation)

Add a new Section 2.1.2, (p. 6):

2.1.2 16 Bit Floating-Point

A 16 bit floating-point number has 1 sign bit (s), 5 exponent bits (e), and 10 mantissa bits (m). The value (v) of a 16 bit floating-point number is determined by the following pseudo code:

```
if (e != 0)
    v = (-1)^s * 2^(e-15) * 1.m  # normalized
else if (f == 0)
    v = (-1)^s * 0               # zero
else
    v = (-1)^s * 2^(e-14) * 0.m  # denormalized
```

It is acceptable for an implementation to treat denormalized 16 bit floating-point numbers as zero.

There are no NAN or infinity values for 16 bit floating-point.

Additions to Chapter 3 of the OpenGL 1.3 Specification (Rasterization)

Section 3.8.1, (p. 116), change the last sentence on the page to:

Each R, G, B, and A value so generated is clamped based on the component type in the <internalFormat>. Fixed-point components are clamped to [0, 1]. Floating-point components are clamped to the limits of the range representable by their format. 32 bit floating- point components are in the standard IEEE float format. 16 bit floating-point components have 1 sign bit, 5 exponent bits, and 10 mantissa bits.

Section 3.8.1, (p. 119), add the following to table 3.16:

Sized Internal Format	Base Internal Format	R bits	G bits	B bits	A bits	L bits	I bits
RGBA_FLOAT32_ATI	RGBA	f32	f32	f32	f32		
RGB_FLOAT32_ATI	RGB	f32	f32	f32			
ALPHA_FLOAT32_ATI	ALPHA				f32		
INTENSITY_FLOAT32_ATI	INTENSITY						f32
LUMINANCE_FLOAT32_ATI	LUMINANCE					f32	
LUMINANCE_ALPHA_FLOAT32_ATI	LUMINANCE_ALPHA				f32	f32	
RGBA_FLOAT16_ATI	RGBA	f16	f16	f16	f16		
RGB_FLOAT16_ATI	RGB	f16	f16	f16			
ALPHA_FLOAT16_ATI	ALPHA				f16		
INTENSITY_FLOAT16_ATI	INTENSITY						f16
LUMINANCE_FLOAT16_ATI	LUMINANCE					f16	
LUMINANCE_ALPHA_FLOAT16_ATI	LUMINANCE_ALPHA				f16	f16	

Additions to Chapter 4 of the OpenGL 1.3 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the OpenGL 1.3 Specification (Special Functions)

None

Additions to Chapter 6 of the OpenGL 1.3 Specification (State and State Requests)

None

Errors

None

New State

None

New Implementation Dependent State

None

Revision History

```
Date: 12/4/2002
Revision: 4
   - Added Section 2.1.2 16 Bit Floating-Point.

Date: 9/11/2002
Revision: 3
   - Changed description of float clamping to be consistent with
     WGL_ATI_pixel_format_float.
```

```
    Date: 9/6/2002
    Revision: 2
       - Changed unsigned integer components to fixed-point components.
       - Resolved GL_FLOAT16_ATI issue.
       - Cleaned up typos.

    Date: 8/18/2002
    Revision: 1
       - First draft for circulation.
```

Name

    ATI_texture_mirror_once

Name Strings

    GL_ATI_texture_mirror_once

Version

    Last Modified Date: 11/14/2000 Revision: 0.30

Number

    221

Dependencies

    EXT_texture3D

Overview

    ATI_texture_mirror_once extends the set of texture wrap modes to
    include two modes (GL_MIRROR_CLAMP_ATI, GL_MIRROR_CLAMP_TO_EDGE_ATI)
    that effectively use a texture map twice as large as the original image
    in which the additional half of the new image is a mirror image of the
    original image.

    This new mode relaxes the need to generate images whose opposite edges
    match by using the original image to generate a matching "mirror image".
    This mode allows the texture to be mirrored only once in the negative
    s, t, and r directions.

Issues

    None known

New Procedure and Functions

    None

New Tokens

    Accepted by the <param> parameter of TexParameteri and TexParameterf,
    and by the <params> parameter of TexParameteriv and TexParameterfv, when
    their <pname> parameter is TEXTURE_WRAP_S, TEXTURE_WRAP_T, or
    TEXTURE_WRAP_R_EXT:

```
      MIRROR_CLAMP_ATI                     0x8742
      MIRROR_CLAMP_TO_EDGE_ATI             0x8743
```

Additions to Chapter 2 of the OpenGL 1.2.1 Specification (Operation)

    None

Additions to Chapter 3 if the OpenGL 1.2.1 Specification (Rasterization):

```
  - (3.8.3, p. 124) Change first three entries in table:

    "TEXTURE_WRAP_S     integer     CLAMP, CLAMP_TO_EDGE, REPEAT,
                                    MIRROR_CLAMP_ATI, MIRROR_CLAMP_TO_EDGE_ATI
     TEXTURE_WRAP_T     integer     CLAMP, CLAMP_TO_EDGE, REPEAT,
                                    MIRROR_CLAMP_ATI, MIRROR_CLAMP_TO_EDGE_ATI
     TEXTURE_WRAP_R     integer     CLAMP, CLAMP_TO_EDGE, REPEAT,
                                    MIRROR_CLAMP_ATI, MIRROR_CLAMP_TO_EDGE_ATI"

  - (3.8.4, p. 125) Added after second paragraph:

    "If TEXTURE_WRAP_S, TEXTURE_WRAP_T, or TEXTURE_WRAP_R_EXT is set to
     MIRROR_CLAMP_ATI or MIRROR_CLAMP_TO_EDGE_ATI, the s (or t or r)
     coordinate is clamped to [-1, 1] and then converted to:

         s      0  <= s <= 1
        -s     -1  <= s <  0

     Like the CLAMP wrap mode, with MIRROR_CLAMP_ATI the texels from
     the border can be used by the texture filter.  MIRROR_CLAMP_TO_EDGE_ATI
     clamps texture coordinates at all mipmap levels such that the texture
     filter never samples a border texel."

  - (3.8.5, p.127) Change last paragraph to:

    "When TEXTURE_MIN_FILTER is LINEAR, a 2 x 2 x 2 cube of texels in the
     image array of level TEXTURE_BASE_LEVEL is selected.  This cube is
     obtained by first clamping texture coordinates as described above
     under Texture Wrap Modes (if the wrap mode for a coordinate is CLAMP,
     CLAMP_TO_EDGE, MIRROR_CLAMP_ATI, or MIRROR_CLAMP_TO_EDGE_ATI) and
     computing..."
```

Additions to Chapter 4:

None

Additions to Chapter 5:

None

Additions to Chapter 6:

None

Additions to the GLX Specification

None

GLX Protocol

None

Errors

None

Dependencies on EXT_texture3D

If EXT_texture3D is not implemented, then the references to clamping of 3D textures in this file are invalid, and references to TEXTURE_WRAP_R_EXT should be ignored.

New State

Only the type information changes for these parameters:

Get Value	Get Command	Type	Initial Value	Attrib
TEXTURE_WRAP_S	GetTexParameteriv	n x Z5	REPEAT	texture
TEXTURE_WRAP_T	GetTexParameteriv	n x Z5	REPEAT	texture
TEXTURE_WRAP_R_EXT	GetTexParameteriv	n x Z5	REPEAT	texture

New Implementation Dependent State

None

Name

    EXT_abgr

Name Strings

    GL_EXT_abgr

Version

    $Date: 1995/03/31 04:40:18 $ $Revision: 1.10 $

Number

    1

Dependencies

    None

Overview

    EXT_abgr extends the list of host-memory color formats. Specifically,
    it provides a reverse-order alternative to image format RGBA. The ABGR
    component order matches the cpack Iris GL format on big-endian machines.

New Procedures and Functions

    None

New Tokens

    Accepted by the <format> parameter of DrawPixels, GetTexImage,
    ReadPixels, TexImage1D, and TexImage2D:

        ABGR_EXT   0x8000

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

    None

Additions to Chapter 3 of the GL Specification (Rasterization)

One entry is added to table 3.5 (DrawPixels and ReadPixels formats). The new table is:

Name	Type	Elements	Target Buffer
COLOR_INDEX	Index	Color Index	Color
STENCIL_INDEX	Index	Stencil value	Stencil
DEPTH_COMPONENT	Component	Depth value	Depth
RED	Component	R	Color
GREEN	Component	G	Color
BLUE	Component	B	Color
ALPHA	Component	A	Color
RGB	Component	R, G, B	Color
RGBA	Component	R, G, B, A	Color
LUMINANCE	Component	Luminance value	Color
LUMINANCE_ALPHA	Component	Luminance value, A	Color
ABGR_EXT	Component	A, B, G, R	Color

Table 3.5: DrawPixels and ReadPixels formats. The third column gives a description of and the number and order of elements in a group.

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

The new format is added to the discussion of Obtaining Pixels from the Framebuffer. It should read " If the <format> is one of RED, GREEN, BLUE, ALPHA, RGB, RGBA, ABGR_EXT, LUMINANCE, or LUMINANCE_ALPHA, and the GL is in color index mode, then the color index is obtained."

The new format is added to the discussion of Index Lookup. It should read "If <format> is one of RED, GREEN, BLUE, ALPHA, RGB, RGBA, ABGR_EXT, LUMINANCE, or LUMINANCE_ALPHA, then the index is used to reference 4 tables of color components: PIXEL_MAP_I_TO_R, PIXEL_MAP_I_TO_G, PIXEL_MAP_I_TO_B, and PIXEL_MAP_I_TO_A."

Additions to Chapter 5 of the GL Specification (Special Functions)

None

Additions to Chapter 6 of the GL Specification (State and State Requests)

None

Additions to the GLX Specification

None

GLX Protocol

One entry is added to tables 1 and 5 in the GLX Protocol Specification:

format	encoding
GL_ABGR_EXT	0x8000

    Table A.2 is also extended:

format	nelements
GL_ABGR_EXT	4

Errors

    None

New State

    None

New Implementation Dependent State

    None

Name

EXT_bgra

Name Strings

GL_EXT_bgra

Version

```
Microsoft revision 1.0, May 19, 1997 (drewb)
$Date: 1997/09/22 23:03:13 $ $Revision: 1.1 $
```

Number

129

Dependencies

None

Overview

EXT_bgra extends the list of host-memory color formats. Specifically, it provides formats which match the memory layout of Windows DIBs so that applications can use the same data in both Windows API calls and OpenGL pixel API calls.

New Procedures and Functions

None

New Tokens

Accepted by the <format> parameter of DrawPixels, GetTexImage, ReadPixels, TexImage1D, and TexImage2D:

```
    BGR_EXT                 0x80E0
    BGRA_EXT                0x80E1
```

Additions to Chapter 2 of the 1.1 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.1 Specification (Rasterization)

One entry is added to table 3.5 (DrawPixels and ReadPixels formats). The new table is:

Name	Type	Elements	Target Buffer
COLOR_INDEX	Index	Color Index	Color
STENCIL_INDEX	Index	Stencil value	Stencil
DEPTH_COMPONENT	Component	Depth value	Depth
RED	Component	R	Color
GREEN	Component	G	Color
BLUE	Component	B	Color
ALPHA	Component	A	Color

RGB	Component	R, G, B	Color
RGBA	Component	R, G, B, A	Color
LUMINANCE	Component	Luminance value	Color
LUMINANCE_ALPHA	Component	Luminance value, A	Color
BGR_EXT	Component	B, G, R	Color
BGRA_EXT	Component	B, G, R, A	Color

Table 3.5: DrawPixels and ReadPixels formats. The third column gives a description of and the number and order of elements in a group.

Additions to Chapter 4 of the 1.1 Specification (Per-Fragment Operations and the Framebuffer)

The new format is added to the discussion of Obtaining Pixels from the Framebuffer. It should read " If the <format> is one of RED, GREEN, BLUE, ALPHA, RGB, RGBA, BGR_EXT, BGRA_EXT, LUMINANCE, or LUMINANCE_ALPHA, and the GL is in color index mode, then the color index is obtained."

The new format is added to the discussion of Index Lookup. It should read "If <format> is one of RED, GREEN, BLUE, ALPHA, RGB, RGBA, BGR_EXT, BGRA_EXT, LUMINANCE, or LUMINANCE_ALPHA, then the index is used to reference 4 tables of color components: PIXEL_MAP_I_TO_R, PIXEL_MAP_I_TO_G, PIXEL_MAP_I_TO_B, and PIXEL_MAP_I_TO_A."

Additions to Chapter 5 of the GL Specification (Special Functions)

None

Additions to Chapter 6 of the GL Specification (State and State Requests)

None

Revision History

Original draft, revision 0.9, October 13, 1995 (drewb)
    Created
Minor revision, revision 1.0, May 19, 1997 (drewb)
    Removed Microsoft Confidential.

Name

    EXT_bindable_uniform

Name String

    GL_EXT_bindable_uniform

Contact

Pat Brown, NVIDIA (pbrown 'at' nvidia.com)
Barthold Lichtenbelt, NVIDIA (blichtenbelt 'at' nvidia.com)

Status

    Shipping for GeForce 8 Series (November 2006)

Version

```
    Last Modified Date:         12/13/2007
    Author revision:            13
```

Number

    342

Dependencies

OpenGL 1.1 is required.

This extension is written against the OpenGL 2.0 specification and version 1.10.59 of the OpenGL Shading Language specification.

This extension interacts with GL_EXT_geometry_shader4.

Overview

This extension introduces the concept of bindable uniforms to the OpenGL Shading Language. A uniform variable can be declared bindable, which means that the storage for the uniform is not allocated by the compiler/linker anymore, but is backed by a buffer object. This buffer object is bound to the bindable uniform through the new command UniformBufferEXT(). Binding needs to happen after linking a program object.

Binding different buffer objects to a bindable uniform allows an application to easily use different "uniform data sets", without having to re-specify the data every time.

A buffer object can be bound to bindable uniforms in different program objects. If those bindable uniforms are all of the same type, accessing a bindable uniform in program object A will result in the same data if the same access is made in program object B. This provides a mechanism for 'environment uniforms', uniform values that can be shared among multiple program objects.

New Procedures and Functions

```
    void UniformBufferEXT(uint program, int location, uint buffer);
    int GetUniformBufferSizeEXT(uint program, int location);
    intptr GetUniformOffsetEXT(uint program, int location);
```

New Tokens

Accepted by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

```
        MAX_VERTEX_BINDABLE_UNIFORMS_EXT                0x8DE2
        MAX_FRAGMENT_BINDABLE_UNIFORMS_EXT              0x8DE3
        MAX_GEOMETRY_BINDABLE_UNIFORMS_EXT              0x8DE4
        MAX_BINDABLE_UNIFORM_SIZE_EXT                   0x8DED
        UNIFORM_BUFFER_BINDING_EXT                      0x8DEF
```

Accepted by the <target> parameters of BindBuffer, BufferData, BufferSubData, MapBuffer, UnmapBuffer, GetBufferSubData, and GetBufferPointerv:

```
        UNIFORM_BUFFER_EXT                              0x8DEE
```

Additions to Chapter 2 of the OpenGL 2.0 Specification (OpenGL Operation)

Modify section 2.15.3 "Shader Variables", page 75.

Add the following paragraph between the second and third paragraph on page 79, "Uniform Variables"

Uniform variables can be further characterized into bindable uniforms. Storage for bindable uniforms does not come out of the, potentially limited, uniform variable storage discussed in the previous paragraph. Instead, storage for a bindable uniform is provided by a buffer object that is bound to the uniform variable. Binding different buffer objects to a bindable uniform allows an application to easily use different "uniform data sets", without having to re-specify the data every time. A buffer object can be bound to bindable uniforms in different program objects. If those bindable uniforms are all of the same type, accessing a bindable uniform in program object A will result in the same data if the same access is made in program object B. This provides a mechanism for 'environment', uniform values that can be shared among multiple program objects.

Change the first sentence of the third paragraph, p. 79, as follows:

When a program object is successfully linked, all non-bindable active uniforms belonging to the program object are initialized to zero (FALSE for Booleans). All active bindable uniforms have their buffer object bindings reset to an invalid state. A successful link will also generate a location for each active uniform, including active bindable uniforms. The values of active uniforms can be changed using this location and the appropriate Uniform* command (see below). For bindable uniforms, a buffer object has to be first bound to the uniform before changing its value. These locations are invalidated.

Change the second to last paragraph, p. 79, as follows:

A valid name for a non-bindable uniform cannot be a structure, an array of structures, or any portion of a single vector or a matrix. A valid name for a bindable uniform cannot be any portion of a single vector or matrix. In order to identify a valid name, ...

Change the fifth paragraph, p. 81, as follows:

The given values are loaded into the uniform variable location identified by <location>. The parameter <location> cannot identify a bindable uniform structure or a bindable uniform array of structures. When loading data for a bindable uniform, the data will be stored in the appropriate location of the buffer object bound to the bindable uniform (see UniformBufferEXT below).

Add the following bullets to the list of errors on p. 82:

- If <location> refers to a bindable uniform structure or a bindable uniform array of structures.

- If <location> refers to a bindable uniform that has no buffer object bound to the uniform.

- If <location> refers to a bindable uniform and the bound buffer object is not of sufficient size. This means that the buffer object is smaller than the size that would be returned by GetUniformBufferSizeEXT for the bindable uniform.

- If <location> refers to a bindable uniform and the buffer object is bound to multiple bindable uniforms in the currently active program object.

Add a sub-section called "Bindable Uniforms" above the section "Samplers", p. 82:

The number of active bindable uniform variables that can be supported by a vertex shader is limited and specified by the implementation dependent constant MAX_VERTEX_BINDABLE_UNIFORMS_EXT. The minimum supported number of bindable uniforms is eight. A link error will be generated if the program object contains more active bindable uniform variables.

To query the minimum size needed for a buffer object to back a given bindable uniform, use the command:

```
  int GetUniformBufferSizeEXT(uint program, int location);
```

This command returns the size in basic machine units of the smallest buffer object that can be used for the bindable uniform given by <location>. The size returned is intended to be passed as the <size> parameter to the BufferData() command. The error INVALID_OPERATION will be generated if <location> does not correspond to an active bindable uniform in <program>. The parameter <location> has to be location corresponding to the name of the bindable uniform itself, otherwise the error INVALID_OPERATION is generated. If the bindable uniform is a structure, <location> can not refer to a structure member. If it is an array, <location> can not refer to any array member other than the first one. If

<program> has not been successfully linked, the error INVALID_OPERATION is generated.

There is an implementation-dependent limit on the size of bindable uniform variables. LinkProgram will fail if the storage required for the uniform (in basic machine units) exceeds MAX_BINDABLE_UNIFORM_SIZE_EXT.

To bind a buffer object to a bindable uniform, use the command:

```
  void UniformBufferEXT(uint program, int location, uint buffer)
```

This command binds the buffer object <buffer> to the bindable uniform <location> in the program object <program>. Any previous binding to the bindable uniform <location> is broken. Before calling UniformBufferEXT the buffer object has to be created, but it does not have to be initialized with data nor its size set. Passing the value zero in <buffer> will unbind the currently bound buffer object. The error INVALID_OPERATION is generated if <location> does not correspond to an active bindable uniform in <program>. The parameter <location> has to correspond to the name of the uniform variable itself, as described for GetUniformBufferSizeEXT, otherwise the error INVALID_OPERATION is generated. If <program> has not been successfully linked, or if <buffer> is not the name of an existing buffer object, the error INVALID_OPERATION is generated.

A buffer object cannot be bound to more than one uniform variable in any single program object. However, a buffer object can be bound to bindable uniform variables in multiple program objects. Furthermore, if those bindable uniforms are all of the same type, accessing a scalar, vector, a member of a structure, or an element of an array in program object A will result in the same data if the same scalar, vector, structure member, or array element is accessed in program object B. Additionally the structures in both program objects have to have the same members, specified in the same order, declared with the same data types and have the same name. If the buffer object bound to the uniform variable is smaller than the minimum size required to store the uniform variable, as reported by GetUniformbufferSizeEXT, the results of reading the variable (or any portion thereof) are undefined.

If LinkProgram is called on a program object that has already been linked, any buffer objects bound to the bindable uniforms in the program are unbound prior to linking, as though UniformBufferEXT were called for each bindable uniform with a <buffer> value of zero.

Buffer objects used to store uniform variables may be created and manipulated by buffer object functions (e.g., BufferData, BufferSubData, MapBuffer) by calling BindBuffer with a <target> of UNIFORM_BUFFER_EXT. It is not necessary to bind a buffer object to UNIFORM_BUFFER_EXT in order to use it with an active program object.

The size and layout of a bindable uniform variable in buffer object storage is not defined. However, the values of signed integer, unsigned integer, or floating-point uniforms may be updated by modifying the underying buffer object storage using either MapBuffer or BufferSubData. The command

```
  intptr GetUniformOffsetEXT(uint program, int location);
```

returns the offset (in bytes) of the uniform in <program> whose location returned by GetUniformLocation is <location>. The error INVALID_VALUE is generated if the object named by <program> does not exist. The error INVALID_OPERATION is generated if <program> is not a program object, if <program> was not linked successfully, or if <location> refers to a uniform that was not declared as bindable. The memory layout of matrix, boolean, or boolean vector uniforms is not defined, and the error INVALID_OPERATION will be generated if <location> refers to a boolean, boolean vector, or matrix uniform. The value -1 is returned by GetUniformOffsetEXT if an error is generated.

The values of such uniforms may be changing by writing signed integer, unsigned integer, or floating-point values into the buffer object at the byte offset returned by GetUniformOffsetEXT. For vectors, two to four integers or floating-point values should be written to consecutive locations in the buffer object storage. For arrays of scalar or vector variables, the number of bytes between individual array members is guaranteed to be constant, but array members are not guaranteed to be stored in adjacent locations. For example, some implementations may pad scalars, or two- or three-component vectors out to a four-component vector.

Change the first paragraph below the sub-heading 'Samplers', p. 82, as follows:

Samplers are special uniforms used in the OpenGL Shading Language to identify the texture object used for each texture lookup. Samplers cannot be declared as bindable in a shader. The value of a sampler indicates the texture image unit being accessed. Setting a sampler's value.

Add the following bullets to the list of error conditions for Begin on p. 87:

- There is one, or more, bindable uniform(s) in the currently active program object that does not have a buffer object bound to it.

- There is one, or more, bindable uniform(s) in the currently active program object that have a buffer object bound to it of insufficient size. This means that the buffer object is smaller than the size that would be returned by GetUniformBufferSizeEXT for the bindable uniform.

- A buffer object is bound to multiple bindable uniforms in the currently active program object.

Additions to Chapter 3 of the OpenGL 2.0 Specification (Rasterization)

Modify Section 3.11.1 "Shader Variables", p. 193

Add a paragraph between the first and second paragraph, p. 194

The number of active bindable uniform variables that can be supported by a fragment shader is limited and specified by the implementation dependent constant MAX_FRAGMENT_BINDABLE_UNIFORMS_EXT. The minimum supported number of bindable uniforms is eight. A link error will be generated if the program object contains more active bindable uniform variables.

Additions to Chapter 4 of the OpenGL 2.0 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 2.0 Specification (Special Functions)

Change section 5.4 Display Lists, p. 237

Add the command UniformBufferEXT to the list of commands that are not compiled into a display list, but executed immediately, under "Program and Shader Objects", p. 241.

Additions to Chapter 6 of the OpenGL 2.0 Specification (State and State Requests)

None.

Additions to Appendix A of the OpenGL 2.0 Specification (Invariance)

None.

Additions to the AGL/GLX/WGL Specifications

None.

Interactions with GL_EXT_geometry_shader4

If GL_EXT_geometry_shader4 is supported, a geometry shader will also support bindable uniforms. The following paragraph needs to be added to the section that discusses geometry shaders:

"The number of active bindable uniform variables that can be supported by a geometry shader is limited and specified by the implementation dependent constant MAX_GEOMETRY_BINDABLE_UNIFORMS_EXT. The minimum supported number of bindable uniforms is eight. A link error will be generated if the program object contains more active bindable uniform variables."

The implementation dependent value MAX_GEOMETRY_BINDABLE_UNIFORMS_EXT will need to be added to the state tables and assigned an enum value.

Errors

The error INVALID_VALUE is generated by UniformBufferEXT, GetUniformBufferSize, or GetUniformOffsetEXT if <program> is not the name of a program or shader object.

The error INVALID_OPERATION is generated by UniformBufferEXT, GetUniformBufferSize, or GetUniformOffsetEXT if <program> is the name of a shader object.

The error INVALID_OPERATION is generated by the Uniform* commands if <location> refers to a bindable uniform structure or an array of such structures.

The error INVALID_OPERATION is generated by the Uniform* commands if

    <location> refers to a bindable uniform that has no buffer object bound.

    The error INVALID_OPERATION is generated by the Uniform* commands if
    <location> refers to a bindable uniform and the bound buffer object is not
    of sufficient size to store data into <location>.

    The error INVALID_OPERATION is generated by the GetUniformBufferSizeEXT
    and UniformBufferEXT commands if <program> has not been successfully
    linked.

    The error INVALID_OPERATION is generated by the GetUniformBufferSizeEXT
    and UniformBufferEXT commands if <location> is not the location
    corresponding to the name of the bindable uniform itself or if <location>
    does not correspond to an active bindable uniform in <program>.

    The error INVALID_OPERATION is generated by GetUniformOffsetEXT if
    <program> was not linked successfully, if <location> refers to a uniform
    that was not declared as bindable, or if <location> refers to a boolean,
    boolean vector, or matrix uniform.

    The error INVALID_OPERATION is generated by the UniformBufferEXT command if
    <buffer> is not the name of a buffer object.

    The error INVALID_OPERATION is generated by Begin, Rasterpos or any
    command that performs an implicit Begin if:

- A buffer object is bound to multiple bindable uniforms in the currently active program object.

- There is one, or more, bindable uniform(s) in the currently active program object that does not have a buffer object bound to it.

- There is one, or more, bindable uniform(s) in the currently active program object that have a buffer object bound to it of insufficient size. This means that the buffer object is smaller than the size that would be returned by GetUniformBufferSizeEXT for the bindable uniform.

New State

Get Value	Type	Get Command	Minimum Value	Description	Section	Attrib
MAX_BINDABLE_VERTEX_UNIFORMS_EXT	Z+	GetIntegerv	8	Number of bindable uniforms per vertex shader	2.15	-
MAX_BINDABLE_FRAGMENT_UNIFORMS_EXT	Z+	GetIntegerv	8	Number of bindable uniforms per fragment shader	3.11.1	-
MAX_BINDABLE_GEOMETRY_UNIFORMS_EXT	Z+	GetIntegerv	8	Number of bindable uniforms per geometry shader	X.X.X	-
MAX_BINDABLE_UNIFORM_SIZE_EXT	Z+	GetIntegerv	16384	Maximum size (in bytes) for bindable uniform storage.	2.15	-

New Implementation Dependent State

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
UNIFORM_BUFFER_BINDING_EXT	Z+	GetIntegerv	0	Uniform buffer bound to the context for buffer object manipulation.	2.15	-

Modifications to The OpenGL Shading Language Specification, Version 1.10.59

Including the following line in a shader can be used to control the language features described in this extension:

```
#extension GL_EXT_bindable_uniform: <behavior>
```

where <behavior> is as specified in section 3.3.

A new preprocessor #define is added to the OpenGL Shading Language:

```
#define GL_EXT_bindable_uniform 1
```

Add to section 3.6 "Keywords"

Add the following keyword:

```
bindable
```

Change section 4.3 "Type Qualifiers"

In the qualifier table, add the following sub-qualifiers under the uniform qualifier:

```
bindable uniform
```

Change section 4.3.5 "Uniform"

Add the following paragraphs between the last and the second to last paragraphs:

Uniform variables, except for samplers, can optionally be further qualified with "bindable". If "bindable" is present, the storage for the uniform comes from a buffer object, which is bound to the uniform through the GL API, as described in section 2.15.3 of the OpenGL 2.0 specification. In this case, the memory used does not count against the storage limit described in the previous paragraph. When using the "bindable" keyword, it must immediately precede the "uniform" keyword.

An example bindable uniform declaration is:

```
bindable uniform float foo;
```

Only a limited number of uniforms can be bindable for each type of shader. If this limit is exceeded, it will cause a compile-time or link-time error. Bindable uniforms that are declared but not used do not count against this limit.

Add to section 9 "Shading Language Grammar"

```
type_qualifer:
   CONST
   ATTRIBUTE  // Vertex only
   uniform-modifieropt UNIFORM

uniform-modifier:
   BINDABLE
```

Issues

1. *Is binding a buffer object to a uniform done before or after linking a program object?*

   DISCUSSION: There is no need to re-link when changing the buffer object that backs a uniform. Re-binding can therefore be relatively quickly. Binding is be done using the location of the uniform retrieved by GetUniformLocation, to make it even faster (instead of binding by name of the uniform).

   Reasons to do this before linking: The linker might want to know what buffer object backs the uniform. Binding of a buffer object to a bindable uniform, in this case, will have to be done using the name of the uniform (no location is available until after linking). Changing the binding of a buffer object to a bindable uniform means the program object will have to be re-linked, which would substantially increase the overhead of using multiple different "constant sets" in a single program.

   RESOLUTION: Binding a buffer object to a bindable uniform needs to be done after the program object is linked. One of the purposes of this extension is to be able to switch among multiple sets of uniform values efficiently.

2. *Is the memory layout of a bindable uniform available to an application?*

   DISCUSSION: Buffer objects are arrays of bytes. The application can map a buffer object and retrieve a pointer to it, and read or write into it directly. Or, the application can use the BufferSubData() command to store data in a buffer object. They can also be filled using ReadPixels (with ARB_pixel_buffer_object), or filled using extensions such as the new transform feedback extension.

   If the layout of a uniform in buffer object memory is known, these different ways of filling a buffer object could be leveraged. On the other hand, different compiler implementations may want a different packing schemes that may or may not match an end-user's expectations (e.g., all individual uniforms might be stored as vec4's). If only the Uniform*() API were allowed to modify buffer objects, we could completely hide the layout of bindable uniforms. Unfortuantely, that would limit how the buffer object can be linked to other sources of data.

   RESOLUTION: RESOLVED. The memory layout of a bindable uniform variable will not be specified. However, a query function will be added that allows applications to determine the layout and load their buffer object via API's other than Uniform*() accordingly if they choose. Unfortunately, the layout may not be consistent across implementations of this extension.

Providing a better standard set of packing rules is highly desirable, and we hope to design and add such functionality in an extension in the near future.

3. *How is synchronization handled between a program object using a buffer object and updates to the buffer object?*

   DISCUSSION: For example, what happens when a ReadPixels into a buffer object is outstanding, that is bound to a bindable uniform while the program object, containing the bindable uniform, is in use?

   RESOLUTION: UNRESOLVED. It is probably the GL implementation's responsibility to properly synchronize such usages. This issue needs solving for GL_EXT_texture_buffer_object also, and should be consistent.

4. *A limited number of bindable uniforms can exist in one program object. Should this limit be queriable?*

   DISCUSSION: The link operation will fail if too many bindable uniforms are declared and active. Should the limit on the number of active bindable uniforms be queriable by the application?

   RESOLUTION: Yes, this limit is queriable.

5. *Is the limit discussed in the previous issue per shader type?*

   DISCUSSION: Is there a different limit for vertex shader and fragment shaders? Hardware might support different limits. The storage for uniform variables is a limit queriable per shader type, thus it would be nice to be consistent with the existing model.

   RESOLUTION: YES.

6. *Can an application find out programmatically that a uniform is declared as a bindable uniform?*

   DISCUSSION: Using GetActiveUniform() the application can programmatically find out which uniforms are active, what their type and size etc it. Do we need to add a mechanism for an application to find out if an active uniform is a bindable uniform?

   RESOLUTION: UNRESOLVED. To be consistent, the answer should be yes. However, extending GetActiveUniform() is not possible, which means we need a new API command. If we define a new API command, it probably is better to define something like: GetNewActiveUniform(int program, uint index, enum property, void *data); Or alternatively, define new API to query the properties of a uniform per uniform location: GetActiveUniformProperty(int program, int location, enum property, void *data)

7. *What to do when the buffer object bound to a bindable uniform is not big enough to back the uniform or if no buffer object is bound at all?*

DISCUSSION: The size of a buffer object can be changed, after it is bound, by calling BufferData. It is possible that the buffer object isn't sufficiently big enough to back the bindable uniform. This is an issue when loading values for uniforms and when actually rendering. In the case of loading uniforms, should the Uniform* API generate an error? In the case of rendering, should this be a Begin error?

RESOLUTION: RESOLVED. It is a Begin error if a buffer object is too small or no buffer object is bound at all. The Uniform* commands will generate an error in these cases as well.

8. *What restrictions are there on binding a buffer object to more than one bindable uniform?*

DISCUSSION: Can a buffer object be bound to more than one uniform within a program object? No, this does not seem to be a good idea. Can a buffer object be bound to more than one uniform in different program objects? Yes, this is useful functionality to have. If each uniform is also of the same type, then data access in program object A then the same access in program object B results in the same data. In the latter case, if the uniform variables are arrays, must the arrays have the same length declared? No, that is too big of a restriction. The application is responsible for making sure the buffer object is sufficiently sized to provide storage for the largest bindable uniform array.

RESOLUTION: RESOLVED.

9. *It is not allowed to bind a buffer object to more than one bindable uniform in a program object. There are several operations that could be affected by this rule: UseProgram(), the uniform loading commands Uniform*, Begin, RasterPos and any related rendering command. Should each operation generate an error if the rule is violated?*

DISCUSSION: See also issue 7. The UseProgram command could generate an error if the rule is violated. However, it is possible to change the binding of a buffer object to a bindable uniform even after UseProgram has been issued. Thus should the Uniform* commands also check for this? If so, is that going to be a performance burden on uniform loading? Or should it be undefined? Finally, at rendering time violation of this rule will have to be checked. If violated, it seems to make sense to generate an error.

RESOLUTION: RESOLVED. Make violation of the rule a Begin error and a Uniform* error.

10. *How to provide the ability to use bindable uniform arrays (or bindable uniform arrays of structures) where the amount of data can differ based on the buffer object bound to it?*

DISCUSSION: In other words, the size of the bindable uniform is no longer declared in the shader, but determined by the buffer object backing it. This can be achieved through a variety of ways:

```
bindable uniform vec3 foo[1];
```

Where we would allow indexing 'off the end' of the array 'foo', because

    it is backed by a buffer object. The actual size of the array will be
    implicitly inferred from the buffer object bound to it. It'll be the
    shader's responsibility to not index outside the size of the buffer
    object. That in turn means that the layout in buffer object memory of a
    bindable uniform needs to be exposed to the application.

    Or we could support something like:

```
    bindable uniform vec3 foo[100000]; // Some really big number
```

    and make all accesses inside the buffer object bound to "foo" legal.

    Or we could support something like:

```
    bindable uniform float foo[];

    foo[3] = 1.0;
    foo[i]  = .
```

    Where 'i' could be a run-time index.

    RESOLUTION: For now, we will not support this functionality.

11. *Do we want to have bindable namespaces instead of the uniform qualifier "bindable"?*

    DISCUSSION: Something like this:

```
    bindable {
      vec3 blarg;
      int booyah;
    };
```

    where "blarg" and "booyah" can be referred to directly, but are both
    bindable to the same buffer. You can achieve this with bindable uniforms
    stored in structures:

```
    bindable uniform struct {
      vec3 blarg;
      int booyah;
    } foo;
```

    but then have to use "foo.blarg" and "foo.booyah".

    RESOLUTION: Not in this extension. This might be nice programming sugar,
    but not essential. Such a feature may be added in a future extension
    building on this one.

12. *How can an application load data into a bindable uniform?*

    RESOLUTION: See also issue 2. Uniform variables declared as bindable can
    be loaded using the existing Uniform* commands, or data can be loaded in
    the buffer object bound to the uniform using any of the existing
    mechanisms.

13. *Should it be allowed to load data, using the Uniform* commands, into a buffer object that is bound to more than one bindable uniform variable in a program object?*

    DISCUSSION: It is a Begin error to attempt to render in this situation.

    RESOLUTION: Yes, to be consistent with the Begin error, it is also an error to load a value in this case.

14. *Should a buffer object binding point be provided for bindable uniforms?*

    DISCUSSION: All current OpenGL buffer object manipulation functions take a <target> to which a buffer object must be bound. In this extension, buffer objects are bound to uniforms stored in a program, and are not bound directly to the context. So these bindings may not be used to manipulate the

    RESOLUTION: Yes, a new <target> called UNIFORM_BUFFER_EXT is provided.

    The following is a simple example of creating, binding, and populating a buffer object for a bindable uniform named "stuff", which is an array of vec4 values:

```
   GLuint program, buffer;
   GLint location, size;
   GLfloat values;

    // ... compile shaders and link <program>
    location = glGetUniformLocation(program, "stuff");
    size = GetUniformBufferSize(program, location);
    glGenBuffers(1, &buffer);
    glBindBuffer(GL_UNIFORM_BUFFER_EXT, buffer);
    glBufferData(GL_UNIFORM_BUFFER_EXT, size, NULL, STATIC_READ);
    glUniformBufferEXT(program, location, buffer);
    ...
    glUseProgram(program);
    glUniform4fv(location, count, values);
```

Revision History

```
    Rev.    Date    Author    Changes
    ----  --------  --------  -----------------------------------------
     13   12/13/07  pbrown    Minor clarification on what values can be passed
                              to GetUniformBufferSizeEXT and UniformBufferEXT.

     12   12/15/06  pbrown    Documented that the '#extension' token
                              for this extension should begin with "GL_",
                              as apparently called for per convention.

     11      --               Pre-release revisions.
```

Name

    EXT_blend_color

Name Strings

    GL_EXT_blend_color

Version

    $Date: 1995/03/31 04:40:19 $ $Revision: 1.7 $

Number

    2

Dependencies

    None

Overview

    Blending capability is extended by defining a constant color that can
    be included in blending equations. A typical usage is blending two
    RGB images. Without the constant blend factor, one image must have
    an alpha channel with each pixel set to the desired blend factor.

New Procedures and Functions

```
    void BlendColorEXT(clampf red,
                       clampf green,
                       clampf blue,
                       clampf alpha);
```

New Tokens

    Accepted by the <sfactor> and <dfactor> parameters of BlendFunc:

```
        CONSTANT_COLOR_EXT                  0x8001
        ONE_MINUS_CONSTANT_COLOR_EXT        0x8002
        CONSTANT_ALPHA_EXT                  0x8003
        ONE_MINUS_CONSTANT_ALPHA_EXT        0x8004
```

    Accepted by the <pname> parameter of GetBooleanv, GetIntegerv,
    GetFloatv, and GetDoublev:

```
        BLEND_COLOR_EXT                     0x8005
```

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

    None

Additions to Chapter 3 of the GL Specification (Rasterization)

    None

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

The commands that control blending are now BlendFunc and BlendColorEXT. A constant color to be used in the blending equation is specified by BlendColorEXT. The four parameters are clamped to the range [0,1] before being stored. The default value for the constant blending color is (0,0,0,0).

The constant color can be used in both the source and destination blending factors. Four lines are added to table 4.1 and table 4.2:

Value	Blend Factors	
ZERO	(0, 0, 0, 0)	
ONE	(1, 1, 1, 1)	
DST_COLOR	(Rd/Kr, Gd/Kg, Bd/Kb, Ad/Ka)	
ONE_MINUS_DST_COLOR	(1, 1, 1, 1) - (Rd/Kr,Gd/Kg,Bd/Kb,Ad/Ka)	
SRC_ALPHA	(As, As, As, As) / Ka	
ONE_MINUS_SRC_ALPHA	(1, 1, 1, 1) - (As, As, As, As) / Ka	
DST_ALPHA	(Ad, Ad, Ad, Ad) / Ka	
ONE_MINUS_DST_ALPHA	(1, 1, 1, 1) - (Ad, Ad, Ad, Ad) / Ka	
CONSTANT_COLOR_EXT	(Rc, Gc, Bc, Ac)	NEW
ONE_MINUS_CONSTANT_COLOR_EXT	(1, 1, 1, 1) - (Rc, Gc, Bc, Ac)	NEW
CONSTANT_ALPHA_EXT	(Ac, Ac, Ac, Ac)	NEW
ONE_MINUS_CONSTANT_ALPHA_EXT	(1, 1, 1, 1) - (Ac, Ac, Ac, Ac)	NEW
SRC_ALPHA_SATURATE	(f, f, f, 1)	

Table 4.1: Values controlling the source blending function and the source blending values they compute. Ka = 2**m - 1, where m is the number of bits in the A color component. Kr, Kg, and Kb are similarly determined by the number of bits in the R, G, and B color components. f = min(As, 1-Ad) / Ka.

Value	Blend Factors	
ZERO	(0, 0, 0, 0)	
ONE	(1, 1, 1, 1)	
SRC_COLOR	(Rs/Kr, Gs/Kg, Bs/Kb, As/Ka)	
ONE_MINUS_SRC_COLOR	(1, 1, 1, 1) - (Rs/Kr,Gs/Kg,Bs/Kb,As/Ka)	
SRC_ALPHA	(As, As, As, As) / Ka	
ONE_MINUS_SRC_ALPHA	(1, 1, 1, 1) - (As, As, As, As) / Ka	
DST_ALPHA	(Ad, Ad, Ad, Ad) / Ka	
ONE_MINUS_DST_ALPHA	(1, 1, 1, 1) - (Ad, Ad, Ad, Ad) / Ka	
CONSTANT_COLOR_EXT	(Rc, Gc, Bc, Ac)	NEW
ONE_MINUS_CONSTANT_COLOR_EXT	(1, 1, 1, 1) - (Rc, Gc, Bc, Ac)	NEW
CONSTANT_ALPHA_EXT	(Ac, Ac, Ac, Ac)	NEW
ONE_MINUS_CONSTANT_ALPHA_EXT	(1, 1, 1, 1) - (Ac, Ac, Ac, Ac)	NEW

Table 4.2: Values controlling the destination blending function and the destination blending values they compute. Ka = 2**m - 1, where m is the number of bits in the A color component. Kr, Kg, and Kb are similarly determined by the number of bits in the R, G, and B color components.

Rc, Gc, Bc, and Ac are the four components of the constant blending color. These blend factors are not scaled by Kr, Kg, Kb, and Ka, because they are already in the range [0,1].

Additions to Chapter 5 of the GL Specification (Special Functions)

None

Additions to Chapter 6 of the GL Specification (State and State Requests)

None

Additions to the GLX Specification

None

GLX Protocol

A new GL rendering command is added. The following command is sent to the server as part of a glXRender request:

```
BlendColorEXT
    2          20              rendering command length
    2          4096            rendering command opcode
    4          FLOAT32         red
    4          FLOAT32         green
    4          FLOAT32         blue
    4          FLOAT32         alpha
```

Errors

INVALID_OPERATION is generated if BlendColorEXT is called between execution of Begin and the corresponding call to End.

New State

Get Value	Get Command	Type	Initial Value	Attrib
BLEND_COLOR_EXT	GetFloatv	C	(0,0,0,0)	color-buffer

New Implementation Dependent State

None

Name

    EXT_blend_equation_separate

Name Strings

    GL_EXT_blend_equation_separate

Notice

Version

    Date: 12/23/2003 Version 1.0

Status

Shipping as of May 2004 for GeForce6.

Number

299

Dependencies

Written based on the wording of the OpenGL 1.5 specification.

OpenGL 1.4 (or ARB_imaging, or EXT_blend_minmax and/or EXT_blend_subtract) is required for blend equation support.

EXT_blend_func_separate is presumed but not required.

EXT_blend_logic_op interacts with this extension.

Overview

EXT_blend_func_separate introduced separate RGB and alpha blend factors. EXT_blend_minmax introduced a distinct blend equation for combining source and destination blend terms. (EXT_blend_subtract & EXT_blend_logic_op added other blend equation modes.) OpenGL 1.4 integrated both functionalities into the core standard.

While there are separate blend functions for the RGB and alpha blend factors, OpenGL 1.4 provides a single blend equation that applies to both RGB and alpha portions of blending.

This extension provides a separate blend equation for RGB and alpha to match the generality available for blend factors.

IP Status

No known IP issues.

Issues

Why not use ATI_blend_equation_separate?

Apple supports this extension in OS X 10.2 but the extension lacks a specification and, as explained in subsequent issues, the token naming is inconsistent with OpenGL conventions.

What should the token names be?

RESOLVED: Follow the precedent of EXT_blend_equation_separate. For example, GL_BLEND_DST becomes GL_BLEND_DST_RGB and GL_BLEND_DST_ALPHA. So GL_BLEND_EQUATION becomes GL_BLEND_EQUATION_RGB (same value as GL_BLEND_EQUATION) and GL_BLEND_EQUATION_ALPHA.

This is different from the ATI_blend_equation_separate approach which introduces the single name GL_ALPHA_BLEND_EQUATION_ATI (no RGB name is introduced). The existing OpenGL convention (example: ARB_texture_env_combine) is to use _RGB and _ALPHA as a suffix for enumerants, not a prefix.

How should get token values be assigned?

RESOLVED: GL_BLEND_EQUATION_RGB_EXT has the same value as GL_BLEND_EQUATION. See "Compatibility" section.

For compatibility with ATI_blend_equation_separate, GL_BLEND_EQUATION_ALPHA_EXT shares the same value (0x883D) with the ATI_blend_equation_separate's GL_ALPHA_BLEND_EQUATION_ATI token. The GL_BLEND_EQUATION_ALPHA_EXT name uses the suffixing convention (rather than prefixing) for adding _ALPHA addition as done by ARB_texture_env_combine and EXT_blend_func_separate.

New Procedures and Functions

```
void BlendEquationSeparateEXT(enum modeRGB,
                              enum modeAlpha);
```

New Tokens

Accepted by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

```
    BLEND_EQUATION_RGB_EXT             0x8009 (same as BLEND_EQUATION)
    BLEND_EQUATION_ALPHA_EXT           0x883D
```

Additions to Chapter 2 of the 1.5 GL Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.5 GL Specification (Rasterization)

None

Additions to Chapter 4 of the 1.5 GL Specification (Per-Fragment Operations and the Framebuffer)

Replace the "Blend Equation" discussion in section 4.1.7 (Blending) with the following:

"The equations used to control blending are determined by the blend equations. Blend equations are specified with the commands:

```
  void BlendEquation(enum mode);
  void BlendEquationSeparateEXT(enum modeRGB, enum modeAlpha);
```

BlendEquationSeparateEXT arguments modeRGB determines the RGB blend function while modeAlpha determines the alpha blend equation. BlendEquation argument mode determines both the RGB and alpha blend equations. modeRGB and modeAlpha must each be one of FUNC_ADD, FUNC_SUBTRACT, FUNC_REVERSE_SUBTRACT, MIN, or MAX.

Destination (framebuffer) components are taken to be fixed-point values represented according to the scheme in section 2.13.9 (Final Color Processing), as are source (fragment) components. Constant color components are taken to be floating point values. [ed: paragraph unchanged except that floating-point is hyphenated.]

Prior to blending, each fixed-point color component undergoes an implied conversion to floating-point. This conversion must leave the values 0 and 1 invariant. Blending components are treated as if carried out in floating-point. [ed: paragraph unchanged except that floating-point is hyphenated.]

Table 4.blendeq provides the corresponding per-component blend equations for each mode, whether acting on RGB components for modeRGB or the alpha component for modeAlpha.

In the table, the "s" subscript on a color component abbreviation (R, G, B, or A) refers to the source color component for an incoming fragment, the "d" subscript on a color component abbreviation refers to the destination color component at the corresponding framebuffer location, and the "c" subscript on a color component abbreviation refers to the constant blend color component. A color component abbreviation without a subscript refers to the new color component resulting from blending. Additionally, Sr, Sg, Sb, and Sa are the red, green, blue, and alpha components of the source weighting factors determined by the source blend function, and Dr, Dg, Db, and Da are the red, green, blue, and alpha components of the destination weighting factors determined by the destination blend function. Blend functions are described below.

```
    Mode                    RGB components          Alpha component
    ---------------------   ---------------------   ---------------------
    FUNC_ADD                Rc = Rs * Sr + Rd * Dr  Ac = As * Sa + Ad * Da
                            Gc = Gs * Sg + Gd * Dg
                            Bc = Bs * Sb + Bd * Db
    ---------------------   ---------------------   ---------------------
    FUNC_SUBTRACT           Rc = Rs * Sr - Rd * Dr  Ac = As * Sa - Ad * Da
                            Gc = Gs * Sg - Gd * Dg
                            Bc = Bs * Sb - Bd * Db
    ---------------------   ---------------------   ---------------------
    FUNC_REVERSE_SUBTRACT   Rc = Rd * Sr - Rs * Dr  Ac = Ad * Sa - As * Da
                            Gc = Gd * Sg - Gs * Dg
                            Bc = Bd * Sb - Bs * Db
    ---------------------   ---------------------   ---------------------
    MIN                     Rc = min(Rs, Rd)        Ac = min(As, Ad)
                            Gc = min(Gs, Gd)
                            Bc = min(Bs, Bd)
    ---------------------   ---------------------   ---------------------
    MAX                     Rc = max(Rs, Rd)        Ac = max(As, Ad)
                            Gc = max(Gs, Gd)
                            Bc = max(Bs, Bd)
    ---------------------   ---------------------   ---------------------

    Table 4.blendeq:  RGB and alpha blend equations are their
    per-component equations controlling the color components resulting
    from blending for each mode."

    In the "Blending State" paragraph, replace the initial lines with...

    "The state required for blending is two integers for the RGB and alpha
    blend equations, four integer indicating the source and destination
    RGB and alpha blending functions, four floating-point values to store
    the RGBA constant blend color, and a bit indicating whether blending
    is enabled or disabled.  The initial blending equations for RGB and
    alpha are FUNC_ADD. ..."
```

Additions to Chapter 5 of the 1.5 GL Specification (Special Functions)

    None

Additions to Chapter 6 of the 1.5 GL Specification (State and State Requests)

    None

Additions to the GLX Specification

    None

GLX Protocol

    A new GL rendering command is added. The following command is sent
    to the server as part of a glXRender request:

```
        BlendEquationSeparateEXT
            2           12              rendering command length
            2           4228            rendering command opcode
            4           ENUM            modeRGB
            4           ENUM            modeAlpha
```

Dependencies on EXT_blend_logic_op

    If EXT_blend_logic_op and EXT_blend_equation_separate are both
    supported, the logic op blend equation should be supported separately
    for RGB and alpha as with the other blend equation modes.

    And add to the table 4.blendeq this line:

```
    Mode                    RGB components          Alpha component
    ---------------------   ---------------------   ---------------------
    LOGIC_OP                Rc = Rs OP Rd           Ac = As OP Ad
                            Gc = Gs OP Gd
                            Bc = Bs OP Bd
    ---------------------   ---------------------   ---------------------
```

    where OP denotes the logical operation controlled by LogicOp (see
    table 4.2).

    Note: there is no support for a distinct RGB logical operation
    and alpha logical operation (that could be provided by another
    extension).

Errors

    INVALID_ENUM is generated if either the modeRGB or modeAlpha
    parameter of BlendEquationSeparateEXT is not one of FUNC_ADD,
    FUNC_SUBTRACT, FUNC_REVERSE_SUBTRACT, MAX, or MIN.

    INVALID_OPERATION is generated if BlendEquationSeparateEXT
    is executed between the execution of Begin and the corresponding
    execution of End.

New State

```
                                                  Initial
    Get Value                 Get Command  Type  Value     Attribute
    ------------------------  -----------  ----  --------  ------------
    BLEND_EQUATION_RGB_EXT    GetIntegerv  Z     FUNC_ADD  color-buffer
    BLEND_EQUATION_ALPHA_EXT  GetIntegerv  Z     FUNC_ADD  color-buffer
```

    [remove BLEND_EQUATION from the table, add a note "v1.5 BLEND_EQUATON"
    beside BLEND_EQUATION_RGB_EXT to note the legacy name.]

New Implementation Dependent State

    None

Compatibility

The BLEND_EQUATION_RGB_EXT query token has the same value as the legacy BLEND_EQUATION query token. This means querying the legacy BLEND_EQUATION state is identical to querying the RGB blend equation state.

This is a different approach than taken by the EXT_blend_func_separate extension, but matches the approach taken by other "split" OpenGL state such as the SMOOTH_POINT_SIZE_RANGE and ALIASED_POINT_SIZE_RANGE values split from POINT_SIZE_RANGE.

In the EXT_blend_func_separate case, four new token names (BLEND_DST_RGB, BLEND_SRC_RGB, BLEND_DST_ALPHA, and BLEND_DST_RGB) with four new token values (0x80C8, 0x80C9, 0x80CA, and 0x80CB respectively) were added. Querying the legacy BLEND_DST (0x0BE0) and BLEND_RGB (0x0BE1) returns the same value as querying BLEND_SRC_RGB and BLEND_DST_RGB respectively but this was never explicitly documented.

In the case of the point size ranges, SMOOTH_POINT_SIZE_RANGE was given the same value as POINT_SIZE_RANGE (0x0B12) and a single new token ALIASED_POINT_SIZE_RANGE (0x846D).

The point size ranges approach is preferable because it minimizes the confusion about how the legacy name should be treated by implementations because the legacy name shares its value with the new name. This is less prone to confusion by developers and implementers and less effort to implement.

For token value compatibility with ATI_blend_equation_separate, GL_BLEND_EQUATION_ALPHA_EXT shares the same value (0x883D) with the ATI_blend_equation_separate's GL_ALPHA_BLEND_EQUATION_ATI token.

Name

    EXT_blend_func_separate

Name Strings

    GL_EXT_blend_func_separate

Version

    Date: 04/06/1999 Version 1.3

Number

    173

Dependencies

    None

Overview

    Blending capability is extended by defining a function that allows
    independent setting of the RGB and alpha blend factors for blend
    operations that require source and destination blend factors. It
    is not always desired that the blending used for RGB is also applied
    to alpha.

New Procedures and Functions

```
    void BlendFuncSeparateEXT(enum sfactorRGB,
                              enum dfactorRGB,
                              enum sfactorAlpha,
                              enum dfactorAlpha);
```

New Tokens

    Accepted by the <pname> parameter of GetBooleanv, GetIntegerv,
    GetFloatv, and GetDoublev:

```
        BLEND_DST_RGB_EXT                 0x80C8
        BLEND_SRC_RGB_EXT                 0x80C9
        BLEND_DST_ALPHA_EXT               0x80CA
        BLEND_SRC_ALPHA_EXT               0x80CB
```

Additions to Chapter 2 of the 1.2 GL Specification (OpenGL Operation)

    None

Additions to Chapter 3 of the 1.2 GL Specification (Rasterization)

    None

Additions to Chapter 4 of the 1.2 GL Specification (Per-Fragment Operations and the Framebuffer)

The RGB and alpha blend factors are separate. The function BlendFuncSeparateEXT allows the specification of the four factors. Table 4.1 and Table 4.2 are modified as follows:

Value	RGB Factors	Alpha Factors
ZERO	(0, 0, 0)	0
ONE	(1, 1, 1)	1
DST_COLOR	(Rd/Kr, Gd/Kg, Bd/Kb)	Ad/Ka
ONE_MINUS_DST_COLOR	(1-Rd/Kr, 1-Gd/Kg, 1-Bd/Kb)	1-Ad/Ka
SRC_ALPHA	(As/Ka, As/Ka, As/Ka)	As/Ka
ONE_MINUS_SRC_ALPHA	(1-As/Ka, 1-As/Ka, 1-As/Ka)	1-As/Ka
DST_ALPHA	(Ad/Ka, Ad/Ka, Ad/Ka)	Ad/Ka
ONE_MINUS_DST_ALPHA	(1-Ad/Ka, 1-Ad/Ka, 1-Ad/Ka)	1-Ad/Ka
CONSTANT_COLOR	(Rc, Gc, Bc)	Ac
ONE_MINUS_CONSTANT_COLOR	(1-Rc, 1-Gc, 1-Bc)	1-Ac
CONSTANT_ALPHA	(Ac, Ac, Ac)	Ac
ONE_MINUS_CONSTANT_ALPHA	(1-Ac, 1-Ac, 1-Ac)	1-Ac
SRC_ALPHA_SATURATE	(f, f, f)	1

Value	RGB Factors	Alpha Factors
ZERO	(0, 0, 0)	0
ONE	(1, 1, 1)	1
SRC_COLOR	(Rs/Kr, Gs/Kg, Bs/Kb)	As/Ka
ONE_MINUS_SRC_COLOR	(1-Rs/Kr, 1-Gs/Kg, 1-Bs/Kb)	1-As/Ka
SRC_ALPHA	(As/Ka, As/Ka, As/Ka)	As/Ka
ONE_MINUS_SRC_ALPHA	(1-As/Ka, 1-As/Ka, 1-As/Ka)	1-As/Ka
DST_ALPHA	(Ad/Ka, Ad/Ka, Ad/Ka)	Ad/Ka
ONE_MINUS_DST_ALPHA	(1-Ad/Ka, 1-Ad/Ka, 1-Ad/Ka)	1-Ad/Ka
CONSTANT_COLOR	(Rc, Gc, Bc)	Ac
ONE_MINUS_CONSTANT_COLOR	(1-Rc, 1-Gc, 1-Bc)	1-Ac
CONSTANT_ALPHA	(Ac, Ac, Ac)	Ac
ONE_MINUS_CONSTANT_ALPHA	(1-Ac, 1-Ac, 1-Ac)	1-Ac
SRC_ALPHA_SATURATE	(f, f, f)	1

The commands that control blending are

```
void BlendFunc(enum src, enum dst)
void BlendFuncSeparateEXT(enum sfactorRGB, enum dfactorRGB,
                          enum sfactorAlpha, enum dfactorAlpha);
```

The BlendFunc command sets both source factors (RGB and alpha) and destination factors (RGB and alpha) while BlendFuncSeparateEXT sets the RGB factors independently from the alpha factors.

Additions to Chapter 5 of the 1.2 GL Specification (Special Functions)

None

Additions to Chapter 6 of the 1.2 GL Specification (State and State Requests)

The state required is four integers indicating the source and destination blending functions for RGB and alpha. The initial state

    for both source functions is ONE. The initial state for both
    destination functions is ZERO.

Additions to the GLX Specification

    None

GLX Protocol

    A new GL rendering command is added. The following command is sent
    to the server as part of a glXRender request:

```
        BlendFuncSeparateEXT
            2           20              rendering command length
            2           4134            rendering command opcode
            4           ENUM            sfactorRGB
            4           ENUM            dfactorRGB
            4           ENUM            sfactorAlpha
            4           ENUM            dfactorAlpha
```

Errors

    GL_INVALID_ENUM is generated if either sfactorRGB, dfactorRGB,
    sfactorAlpha, or dfactorAlpha is not an accepted value.

    GL_INVALID_OPERATION is generated if glBlendFunc is executed between
    the execution of glBegin and the corresponding execution of glEnd.

New State

    The get values BLEND_SRC and BLEND_DST return the RGB source and
    destination factor, respectively.

```
                                             Initial
    Get Value             Get Command   Type  Value       Attribute
    ---------             -----------   ----  -------     ------------
    BLEND_SRC_RGB_EXT     GetFloatv     Z     ONE        color-buffer
    BLEND_DST_RGB_EXT     GetFloatv     Z     ZERO       color-buffer
    BLEND_SRC_ALPHA_EXT   GetFloatv     Z     ONE        color-buffer
    BLEND_DST_ALPHA_EXT   GetFloatv     Z     ZERO       color-buffer
```

New Implementation Dependent State

    None

Name

    EXT_blend_minmax

Name Strings

    GL_EXT_blend_minmax

Version

    $Date: 1995/03/31 04:40:34 $ $Revision: 1.3 $

Number

    37

Dependencies

    None

Overview

    Blending capability is extended by respecifying the entire blend
    equation. While this document defines only two new equations, the
    BlendEquationEXT procedure that it defines will be used by subsequent
    extensions to define additional blending equations.

    The two new equations defined by this extension produce the minimum
    (or maximum) color components of the source and destination colors.
    Taking the maximum is useful for applications such as maximum projection
    in medical imaging.

Issues

    *   I've prefixed the ADD token with FUNC, to indicate that the blend
        equation includes the parameters specified by BlendFunc. (The min
        and max equations don't.) Is this necessary? Is it too ugly?
        Is there a better way to accomplish the same thing?

New Procedures and Functions

    void BlendEquationEXT(enum mode);

New Tokens

    Accepted by the <mode> parameter of BlendEquationEXT:

        FUNC_ADD_EXT   0x8006
        MIN_EXT   0x8007
        MAX_EXT   0x8008

    Accepted by the <pname> parameter of GetBooleanv, GetIntegerv,
    GetFloatv, and GetDoublev:

        BLEND_EQUATION_EXT   0x8009

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

None

Additions to Chapter 3 of the GL Specification (Rasterization)

None

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

The GL Specification defines a single blending equation. This extension introduces a blend equation mode that is specified by calling BlendEquationEXT with one of three enumerated values. The default value FUNC_ADD_EXT specifies that the blending equation defined in the GL Specification be used. This equation is

```
    C' = (Cs * S) + (Cd * D)

         /  1.0     C' > 1.0
    C = (
         \   C'     C' <= 1.0
```

where Cs and Cd are the source and destination colors, and S and D are as specified by BlendFunc.

If BlendEquationEXT is called with <mode> set to MIN_EXT, the blending equation becomes

```
    C = min (Cs, Cd)
```

Finally, if BlendEquationEXT is called with <mode> set to MAX_EXT, the blending equation becomes

```
    C = max (Cs, Cd)
```

In all cases the blending equation is evaluated separately for each color component.

Additions to Chapter 5 of the GL Specification (Special Functions)

None

Additions to Chapter 6 of the GL Specification (State and State Requests)

None

Additions to the GLX Specification

None

GLX Protocol

A new GL rendering command is added. The following command is sent to the server as part of a glXRender request:

```
        BlendEquationEXT
            2           8               rendering command length
            2           4097            rendering command opcode
            4           ENUM            mode
```

Errors

    INVALID_ENUM is generated by BlendEquationEXT if its single parameter
    is not FUNC_ADD_EXT, MIN_EXT, or MAX_EXT.

    INVALID_OPERATION is generated if BlendEquationEXT is executed between
    the execution of Begin and the corresponding execution to End.

New State

```
    Get Value           Get Command     Type    Initial Value   Attribute
    ---------           -----------     ----    -------------   ---------
    BLEND_EQUATION_EXT  GetIntegerv     Z3      FUNC_ADD_EXT    color-buffer
```

New Implementation Dependent State

    None

Name

    EXT_blend_subtract

Name Strings

    GL_EXT_blend_subtract

Version

    $Date: 1995/03/31 04:40:39 $ $Revision: 1.4 $

Number

    38

Dependencies

    EXT_blend_minmax affects the definition of this extension

Overview

    Two additional blending equations are specified using the interface
    defined by EXT_blend_minmax. These equations are similar to the
    default blending equation, but produce the difference of its left
    and right hand sides, rather than the sum. Image differences are
    useful in many image processing applications.

New Procedures and Functions

    None

New Tokens

    Accepted by the <mode> parameter of BlendEquationEXT:

        FUNC_SUBTRACT_EXT   0x800A
        FUNC_REVERSE_SUBTRACT_EXT   0x800B

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

    None

Additions to Chapter 3 of the GL Specification (Rasterization)

    None

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

```
    Two additional blending equations are defined.  If BlendEquationEXT is
    called with <mode> set to FUNC_SUBTRACT_EXT, the blending equation
    becomes

        C' = (Cs * S) - (Cd * D)

             /  0.0     C' < 0.0
        C = (
             \   C'     C' >= 0.0

    where Cs and Cd are the source and destination colors, and S and D are
    as specified by BlendFunc.

    If BlendEquationEXT is called with <mode> set to
    FUNC_REVERSE_SUBTRACT_EXT, the blending equation becomes

        C' = (Cd * D) - (Cs * S)

             /  0.0     C' < 0.0
        C = (
             \   C'     C' >= 0.0

    In all cases the blending equation is evaluated separately for each
    color component.
```

Additions to Chapter 5 of the GL Specification (Special Functions)

None

Additions to Chapter 6 of the GL Specification (State and State Requests)

None

Additions to the GLX Specification

None

GLX Protocol

None

Dependencies on EXT_blend_minmax

If this extension is supported, but EXT_blend_minmax is not, then this extension effectively defines the procedure BlendEquationEXT, its parameter FUNC_ADD_EXT, and the query target BLEND_EQUATION_EXT, as described in EXT_blend_minmax. It is therefore as though EXT_blend_minmax were also supported, except that equations MIN_EXT and MAX_EXT are not supported.

Errors

```
    INVALID_ENUM is generated by BlendEquationEXT if its single parameter
    is not FUNC_ADD_EXT, MIN_EXT, MAX_EXT, FUNC_SUBTRACT_EXT, or
    FUNC_REVERSE_SUBTRACT_EXT.

    INVALID_OPERATION is generated if BlendEquationEXT is executed between
    the execution of Begin and the corresponding execution to End.
```

New State

```
    Get Value           Get Command     Type    Initial Value   Attribute
    ---------           -----------     ----    -------------   ---------
    BLEND_EQUATION_EXT  GetIntegerv     Z5      FUNC_ADD_EXT    color-buffer
```

New Implementation Dependent State

```
    None
```

Name

    EXT_clip_volume_hint

Name Strings

    GL_EXT_clip_volume_hint

Version

    Microsoft revision 1.00, April 17, 1996 (hockl)

Number

    79

Dependencies

    None.

Overview

    EXT_clip_volume_hint provides a mechanism for applications to
    indicate that they do not require clip volume clipping for
    primitives. It allows applications to maximize performance in
    situations where they know that clipping is unnecessary.
    EXT_clip_volume_hint is only an indication, though, and
    implementations are free to ignore it.

New Procedures and Functions

    None.

New Tokens

```
    Accepted by the target parameter of Hint and the pname parameter of
    GetBooleanv, GetDoublev, GetFloatv and GetIntegerv:
        CLIP_VOLUME_CLIPPING_HINT_EXT   0x80F0
```

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

    Before the last paragraph of Section 2.11, 'Clipping,' the following
    text is added:

        The EXT_clip_volume_hint extension can be used to indicate that
        a primitive falls inside the current clip volume. In this case,
        an implementation might not clip the primitive to the clip
        volume, and the behavior of the GL is undefined if the primitive
        extends beyond the clip volume.

    In the fourth (clipping) paragraph of Section 2.12, 'Current Raster
    Position,' the following text is added before the last sentence
    "Figure 2.7 summarizes..."

        Raster position clipping is not affected by the
        CLIP_VOLUME_CLIPPING_HINT_EXT hint in the EXT_clip_volume_hint
        extension.

Additions to Chapter 3 of the GL Specification (Rasterization)

None.

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

None.

Additions to Chapter 5 of the GL Specification (Special Functions)

Section 5.6, 'Hints,' should be changed to add the following hint description:

CLIP_VOLUME_CLIPPING_HINT_EXT, indicating whether clipping to the clip volume is necessary.

Additions to Chapter 6 of the GL Specification (State and State Requests)

In table 6.18, 'Hints,' the following entry is added:

```
    CLIP_VOLUME_CLIPPING_HINT_EXT|Z3|GetIntegerv|DONT_CARE|
    Clip volume clipping hint|5.6|hint
```

Revision History

```
    Original draft, revision 0.9, March 1, 1996 (drewb)
            Created.
    Minor revision, revision 0.91, March 8, 1996 (drewb)
    Hint revision, revision 0.95, April 12, 1996 (drewb)
        Changed from Enable-based to Hint-based.  Clarified
        behavior of RasterPos.
    More revision, revision 0.96, April 16, 1996 (hockl)
        Changed extension and enumerant names.  Added robustness.
        Changed it to have no effect on RasterPos.
    More revision, revision 1.00, April 17, 1996 (hockl)
        Removed robustness requirement.
```

XXX - Not complete yet!!!

Name

    EXT_compiled_vertex_array

Name Strings

    GL_EXT_compiled_vertex_array

Version

    $Date: 1996/11/21 00:52:19 $ $Revision: 1.3 $

Number

    97

Dependencies

    None

Overview

This extension defines an interface which allows static vertex array data to be cached or pre-compiled for more efficient rendering. This is useful for implementations which can cache the transformed results of array data for reuse by several DrawArrays, ArrayElement, or DrawElements commands. It is also useful for implementations which can transfer array data to fast memory for more efficient processing.

For example, rendering an M by N mesh of quadrilaterals can be accomplished by setting up vertex arrays containing all of the vertexes in the mesh and issuing M DrawElements commands each of which operate on 2 * N vertexes. Each DrawElements command after the first will share N vertexes with the preceding DrawElements command. If the vertex array data is locked while the DrawElements commands are executed, then OpenGL may be able to transform each of these shared vertexes just once.

Issues

* Is compiled_vertex_array the right name for this extension?

* Should there be an implementation defined maximum number of array elements which can be locked at a time (i.e. MAX_LOCKED_ARRAY_SIZE)?

  Probably not, the lock request can always be ignored with no resulting change in functionality if there are insufficent resources, and allowing the GL to define this limit can make things difficult for applications.

* Should there be any restrictions on what state can be changed while the vertex array data is locked?

  Probably not. The GL can check for state changes and invalidate any cached vertex state that may be affected. This is likely to cause a performance hit, so the preferred use will be to not change state while the vertex array data is locked.

New Procedures and Functions

```
    void LockArraysEXT (int first, sizei count)
    void UnlockArraysEXT (void)
```

New Tokens

Accepted by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

```
        ARRAY_ELEMENT_LOCK_FIRST_EXT        0x81A8
        ARRAY_ELEMENT_LOCK_COUNT_EXT        0x81A9
```

Additions to Chapter 2 of the 1.1 Specification (OpenGL Operation)

After the discussion of InterleavedArrays, add a description of array compiling/locking.

The currently enabled vertex arrays can be locked with the command LockArraysEXT. When the vertex arrays are locked, the GL can compile the array data or the transformed results of array data associated with the currently enabled vertex arrays. The vertex arrays are unlocked by the command UnlockArraysEXT.

Between LockArraysEXT and UnlockArraysEXT the application should ensure that none of the array data in the range of elements specified by <first> and <count> are changed. Changes to the array data between the execution of LockArraysEXT and UnlockArraysEXT commands may affect calls may affect DrawArrays, ArrayElement, or DrawElements commands in non-sequential ways.

While using a compiled vertex array, references to array elements by the commands DrawArrays, ArrayElement, or DrawElements which are outside of the range specified by <first> and <count> are undefined.

Additions to Chapter 3 of the 1.1 Specification (Rasterization)

None

Additions to Chapter 4 of the 1.1 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.1 Specification (Special Functions)

LockArraysEXT and UnlockArraysEXT are not complied into display lists but are executed immediately.

Additions to Chapter 6 of the 1.1 Specification (State and State Requests)

None

Additions to the GLX Specification

XXX - Not complete yet!!!

GLX Protocol

XXX - Not complete yet!!!

Errors

INVALID_VALUE is generated if LockArrarysEXT parameter <first> is less than zero.

INVALID_VALUE is generated if LockArraysEXT parameter <count> is less than or equal to zero.

INVALID_OPERATION is generated if LockArraysEXT is called between execution of LockArraysEXT and corresponding execution of UnlockArraysEXT.

INVALID_OPERATION is generated if UnlockArraysEXT is called without a corresponding previous execution of LockArraysEXT.

INVALID_OPERATION is generated if LockArraysEXT or UnlockArraysEXT is called between execution of Begin and the corresponding execution of End.

New State

Get Value	Get Command	Type	Initial Value	Attrib
ARRAY_ELEMENT_LOCK_FIRST_EXT	GetIntegerv	Z+	0	client-vertex-array
ARRAY_ELEMENT_LOCK_COUNT_EXT	GetIntegerv	Z+	0	client-vertex-array

New Implementation Dependent State

None

Name

    EXT_depth_bounds_test

Name Strings

    GL_EXT_depth_bounds_test

Notice

    Copyright NVIDIA Corporation, 2002, 2003.

Status

    Implemented in GeForce FX 5900 (NV35) drivers as of June 2003.

    Also supported by GeForce FX 5700 (NV36) and GeForce6 (NV4x).

Version

    Last Modified Date: $Date: 2004/05/17 $
    NVIDIA Revision: $Revision: #5 $

Number

    297

Dependencies

    Written based on the wording of the OpenGL 1.3 specification.

Overview

This extension adds a new per-fragment test that is, logically, after the scissor test and before the alpha test. The depth bounds test compares the depth value stored at the location given by the incoming fragment's (xw,yw) coordinates to a user-defined minimum and maximum depth value. If the stored depth value is outside the user-defined range (exclusive), the incoming fragment is discarded.

Unlike the depth test, the depth bounds test has NO dependency on the fragment's window-space depth value.

This functionality is useful in the context of attenuated stenciled shadow volume rendering. To motivate the functionality's utility in this context, we first describe how conventional scissor testing can be used to optimize shadow volume rendering.

If an attenuated light source's illumination can be bounded to a rectangle in XY window-space, the conventional scissor test can be used to discard shadow volume fragments that are guaranteed to be outside the light source's window-space XY rectangle. The stencil increments and decrements that would otherwise be generated by these scissored fragments are inconsequential because the light source's illumination can pre-determined to be fully attenuated outside the scissored region. In other words, the scissor test can be used to discard shadow volume fragments rendered outside the scissor, thereby improving performance, without affecting the ultimate illumination of these pixels with respect to the attenuated light source.

This scissoring optimization can be used both when rendering the stenciled shadow volumes to update stencil (incrementing and decrementing the stencil buffer) AND when adding the illumination contribution of attenuated light source's.

In a similar fashion, we can compute the attenuated light source's window-space Z bounds (zmin,zmax) of consequential illumination. Unless a depth value (in the depth buffer) at a pixel is within the range [zmin,zmax], the light source's illumination can be pre-determined to be inconsequential for the pixel. Said another way, the pixel being illuminated is either far enough in front of or behind the attenuated light source so that the light source's illumination for the pixel is fully attenuated. The depth bounds test can perform this test.

Issues

Where should the depth bounds test take place in the OpenGL fragment processing pipeline?

RESOLUTION: After scissor test, before alpha test. In practice, this is a logical placement of the test. An implementation is free to perform the test in a manner that is consistent with the specified ordering.

Importantly, the depth bounds test occurs before any fragment operation that has a side-effect such as stencil and/or depth buffer writes (ie, the stencil or depth test). This makes it possible to discard incoming fragment's without concern for preserving such side-effects.

Is the depth bounds test consistent with early depth rejection?

Yes. If an OpenGL implementation supports some conservative bounds on depth values in subregions of the depth buffer (hierarchical depth buffers, etc), the depth bounds test can reject fragments based on these conservative bounds.

How are the depth bounds specified?

RESOLUTION: Normalized window-space depth values. This means the depth values are specified in the range [0.0, 1.0] similar to glDepthRange.

Can the zmin bound be greater than the zmax bound?

RESOLUTION: zmin must be less than or equal to zmax or an INVALID_VALUE error is generated.

Another way to interpret this situation is to have zmin>zmax reject all fragments where the corresponding pixel's depth value is between zmin and zmax. But this does not seem useful enough to specify.

What should the glDepthBoundsEXT routine mimic?

RESOLUTION: glDepthBoundsEXT should mimic glDepthRange in parameter types and clamping, excepting that zmin must be less than zmax.

Do the depth bounds have anything to do with the depth range?

RESOLUTION: No. These are totally independent pieces of state. To reinforce the point, having a depth range and depth bounds with no overlap is perfectly well-defined (even if a little odd).

What push/pop attrib bits should affect the depth bounds test enable?

RESOLUTION: GL_ENABLE_BIT and GL_DEPTH_BUFFER_BIT.

How does depth bounds testing interact with polygon offset or depth replace operations (say from ARB_fragment_program, NV_texture_shader, or NV_fragment_program)?

RESOLUTION: There are NO interactions. The depth bounds test has NO dependency on the incoming fragment's depth value so it doesn't matter if there is a polygon offset or depth replace operation.

Does depth bounds testing affect bitmap/draw/copy pixels operations involving depth component pixels?

RESOLUTION: Yes, depth bounds testing affects all rasterized primitives (just like all other fragment operations).

How does depth bounds test interact with multisampling?

RESOLUTION: The depth bounds test is performed per-sample when multisampling is active, just like the depth test.

At what precision is the depth bounds test carried out?

RESOLUTION: For the purposes of the test, the bounds are converted to fixed-point as though they were to be written to the depth buffer, and the comparison uses those quantized bounds.

Can you have the depth test disabled and still have the depth bounds test enabled?

RESOLUTION: Yes. The two tests operate independently.

How does the depth bounds test operate if there is no depth buffer?

RESOLUTION: It is as if the depth bounds test always passes (analogous to the depth test).

New Procedures and Functions

```
void DepthBoundsEXT(clampd zmin, clampd zmax);
```

New Tokens

    Accepted by the <cap> parameter of Enable, Disable, and IsEnabled,
    and by the <pname> parameter of GetBooleanv, GetIntegerv,
    GetFloatv, and GetDoublev:

        DEPTH_BOUNDS_TEST_EXT   0x8890

    Accepted by the <pname> parameter of GetBooleanv, GetIntegerv,
    GetFloatv, and GetDoublev:

        DEPTH_BOUNDS_EXT   0x8891

Additions to Chapter 2 of the OpenGL 1.3 Specification (OpenGL Operation)

    None

Additions to Chapter 3 of the OpenGL 1.3 Specification (Rasterization)

    None

Additions to Chapter 4 of the OpenGL 1.3 Specification (Per-Fragment Operations and the Framebuffer)

-- Figure 4.1 Per-fragment operations

    Add a block for the "depth bounds test" after the scissor and before
    the alpha test.

-- Section 4.1.X Depth Bounds Test (following Section 4.1.2 Scissor Test)

    "The depth bounds test determines whether the depth value (Zpixel)
    stored at the location given by the incoming fragment's (xw,yw)
    location lies within the depth bounds range defined by two values.
    These values are set with

        void DepthBoundsEXT(clampd zmin, clampd zmax);

    Each of zmin and zmax are clamped to lie within [0,1] (being of
    type clampd). If zmin <= Zpixel <= zmax, then the depth bounds test
    passes. Otherwise, the test fails and the fragment is discarded.
    The test is enabled or disabled using Enable or Disable using the
    constant DEPTH_BOUNDS_TEST_EXT. When disabled, it is as if the depth
    bounds test always passes. If zmin is greater than zmax, then the
    error INVALID_VALUE is generated. The state required consists of
    two floating-point values and a bit indicating whether the test is
    enabled or disabled. In the initial state, zmin and zmax are set
    to 0.0 and 1.0 respectively; and the depth bounds test is disabled.

    If there is no depth buffer, it is as if the depth bounds test always
    passes."

-- Section 4.10 Additional Multisample Fragment Operations

    Add depth bounds test to the list of operations affected by
    multisampling. Amend the 1st and 2nd sentences in the 2nd paragraph
    to read:

    "If MULTISAMPLE is enabled, and the value of SAMPLE_BUFFERS is one,
    the depth bounds test, alpha test, depth test, blending, and dithering
    operations are performed for each pixel sample, rather than just once
    for each fragment. Failure of the depth bounds, alpha, stencil, or
    depth test results in termination of the processing of the sample,
    rather than discarding of the fragment."

    Amend the 1st sentence in the 3nd paragraph to read:

    "Depth bounds, stencil, depth, blending, and dithering operations
    are performed for a pixel sample only if that sample's fragment
    coverage bit is a value of 1."

    Amend the 3rd sentence in the 4th paragraph to read:

    "An implementation may choose to identify a centermost sample, and
    to perform depth bounds, alpha, stencil, and depth tests on only
    that sample."

Additions to Chapter 5 of the OpenGL 1.3 Specification (Special Functions)

    None

Additions to Chapter 6 of the OpenGL 1.3 Specification (State and State Requests)

    None

Additions to the AGL/GLX/WGL Specifications

    None

GLX Protocol

    A new GL rendering command is added. The following command is sent to the
    server as part of a glXRender request:

```
        DepthBoundsEXT
            2           12              rendering command length
            2           4229            rendering command opcode
            4           FLOAT32         zmin
            4           FLOAT32         zmax
```

Errors

    If zmin is greater than zmax, then the error INVALID_VALUE is
    generated.

New State

(table 6.15 "Pixel Operation)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
DEPTH_BOUNDS_TEST_EXT	B	IsEnabled	False	Depth bounds test enable	4.1.X	depth-buffer/enable
DEPTH_BOUNDS_EXT	2xR+	GetFloatv	0,1	Depth bounds zmin & zmax	4.1.X	depth-buffer

New Implementation Dependent State

None

Revision History

NVIDIA exposed a functionally and enumerant identical version of this extension under the name NV_depth_bounds_test. NVIDIA drivers after May 2003 support the EXT_depth_bounds_test name only.

Mesa and NVIDIA agreed to make this an EXT extension in April 2003.

8/27/2003 - GLX protocol specification added.

Name

    EXT_draw_buffers2

Name Strings

    GL_EXT_draw_buffers2

Contact

    Mike Strauss, NVIDIA Corporation (mstrauss 'at' nvidia.com)

Status

    Shipping for GeForce 8 Series (November 2006)

Version

```
    Last Modified Date:         11/06/2006
    NVIDIA Revision:            9
```

Number

    340

Dependencies

The extension is written against the OpenGL 2.0 Specification.

OpenGL 2.0 is required.

Overview

This extension builds upon the ARB_draw_buffers extension and provides separate blend enables and color write masks for each color output. In ARB_draw_buffers (part of OpenGL 2.0), separate values can be written to each color buffer, but the blend enable and color write mask are global and apply to all color outputs.

While this extension does provide separate blend enables, it does not provide separate blend functions or blend equations per color output.

New Procedures and Functions

```
    void ColorMaskIndexedEXT(uint buf, boolean r, boolean g,
                             boolean b, boolean a);

    void GetBooleanIndexedvEXT(enum value, uint index, boolean *data);

    void GetIntegerIndexedvEXT(enum value, uint index, int *data);

    void EnableIndexedEXT(enum target, uint index);

    void DisableIndexedEXT(enum target, uint index);

    boolean IsEnabledIndexedEXT(enum target, uint index);
```

New Tokens

    None.

Additions to Chapter 2 of the OpenGL 2.0 Specification (OpenGL Operation)

    None.

Additions to Chapter 3 of the OpenGL 2.0 Specification (Rasterization)

    None.

Additions to Chapter 4 of the OpenGL 2.0 Specification (Per-Fragment Operations and the Frame Buffer)

Modify the thrid paragraph of section 4.1.8 (Blending) , p206, to read as follows:

Blending is dependent on the incoming fragment's alpha value and that of the corresponding currently stored pixel. Blending applies only in RGBA mode; in color index mode it is bypassed. Blending is enabled or disabled for an individual draw buffer using

```
void EnableIndexedEXT(GLenum target, GLuint index);
void DisableIndexedEXT(GLenum target, GLuint index);
```

<target> is the symbolic constant BLEND and <index> is an integer i specifying the draw buffer associated with the symbolic constant DRAW_BUFFERi. If the color buffer associated with DRAW_BUFFERi is one of FRONT, BACK, LEFT, RIGHT, or FRONT_AND_BACK (specifying multiple color buffers), then the state enabled or disabled is applicable for all of the buffers. Blending can be enabled or disabled for all draw buffers using Enable or Disable with the symbolic constant BLEND. If blending is disabled for a particular draw buffer, or if logical operation on color values is enabled (section 4.1.10), proceed to the next operation.

Modify the first paragraph of section 4.1.8 (Blending - Blending State), p209, to read as follows:

The state required for blending is two integers for the RGB and alpha blend equations, four integers indicating the source and destination RGB and alpha blending functions, four floating-point values to store the RGBA constant blend color, and n bits indicating whether blending is enabled or disabled for each of the n draw buffers. The initial blend equations for RGB and alpha are both FUNC_ADD. The initial blending functions are ONE for the source RGB and alpha functions, and ZERO for the destination RGB and alpha functions. The initial constant blend color is (R, G, B, A) = (0, 0, 0, 0). Initially, blending is disabled for all draw buffers.

Modify the first paragraph of section 4.2.2 (Fine Control of Buffer Updates) to read as followS:

Three commands are used to mask the writing of bits to each of the logical draw buffers after all per-fragment operations have been performed.

The commands

```
    void IndexMask(uint mask);
    void ColorMask(boolean r, boolean g, boolean b, boolean a);
    void ColorMaskIndexedEXT(uint buf, boolean r, boolean g,
                             boolean b, boolean a);
```

control writes to the active draw buffers.

The least significant n bits of <mask>, where n is the number of bits in a color index buffer, specify a mask. Where a 1 appears in this mask, the corresponding bit in the color index buffer (or buffers) is written; where a 0 appears, the bit is not written. This mask applies only in color index mode.

In RGBA mode, ColorMask and ColorMaskIndexedEXT are used to mask the writing of R, G, B and A values to the draw buffer or buffers. ColorMaskIndexedEXT sets the mask for a particular draw buffer. The mask for DRAW_BUFFERi is modified by passing i as the parameter <buf>. <r>, <g>, <b>, and <a> indicate whether R, G, B, or A values, respectively, are written or not (a value of TRUE means that the corresponding value is written). The mask specified by <r>, <g>, <b>, and <a> is applied to the color buffer associated with DRAW_BUFFERi. If DRAW_BUFFERi is one of FRONT, BACK, LEFT, RIGHT, or FRONT_AND_BACK (specifying multiple color buffers) then the mask is applied to all of the buffers. ColorMask sets the mask for all draw buffers to the same values as specified by <r>, <g>, <b>, and <a>.

Additions to Chapter 5 of the OpenGL 2.0 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 2.0 Specification (State and State Requests)

Modify the second paragraph of section 6.1.1 (Simple Queries) p244 to read as follows:

...<data> is a pointer to a scalar or array of the indicated type in which to place the returned data.

```
    void GetBooleanIndexedvEXT(enum target, uint index, boolean *data);
    void GetIntegerIndexedvEXT(enum target, uint index, int *data);
```

are used to query indexed state. <target> is the name of the indexed state and <index> is the index of the particular element being queried. <data> is a pointer to a scalar or array

```
    of the indicated type in which to place the returned data.  In
    addition

        boolean IsEnabled(enum value);

    can be used to determine if <value> is currently enabled (as with
    Enable) or disabled.

        boolean IsEnabledIndexedEXT(enum target, uint index);

    can be used to determine if the index state corresponding to
    <target> and <index> is enabled or disabled.
```

Additions to Appendix A of the OpenGL 2.0 Specification (Invariance)

None.

Additions to the AGL/GLX/WGL Specifications

None.

Errors

The error INVALID_ENUM is generated by EnableIndexedEXT and DisableIndexedEXT if the <target> parameter is not BLEND.

The error INVALID_OPERATION is generated by EnableIndexedEXT and DisableIndexeEXT if the <target> parameter is BLEND and the <index> parameter is outside the range [0, MAX_DRAW_BUFFERS-1].

The error INVALID_ENUM is generated by IsEnabledIndexedEXT if the <target> parameter is not BLEND.

The error INVALID_OPERATION is generated by IsEnabledIndexedEXT if the <target> parameter is BLEND and the <index> parameter is outside the range [0, MAX_DRAW_BUFFERS-1].

The error INVALID_OPERATION is generated by DrawBufferColorMaskEXT if the <buf> parameter is outside the range [0, MAX_DRAW_BUFFERS-1].

The error INVALID_ENUM is generated by GetBooleanIndexedvEXT if the <target> parameter is not BLEND.

The error INVALID_OPERATION is generated by GetBooleanIndexedvEXT if the <target> parameter is BLEND and the <index> parameter is outside the range [0, MAX_DRAW_BUFFERS-1].

New State

    Modify (table 6.20, p281), modifying the entry for BLEND and adding
    a new one.

Get Target	Type	Get Command	Value	Description	Section	Attribute
BLEND	B	IsEnabled	False	Blending enabled for draw buffer 0	4.1.8	color-buffer/enable
BLEND	B	IsEnabledIndexedEXT	False	Blending enabled for draw buffer i where i is specified as <index>	4.1.8	color-buffer/enable

    Modify (table 6.21, p282), modifying the entry for COLOR_WRITEMASK
    and adding a new one.

Get Value	Type	Get Command	Value	Description	Section	Attribute
COLOR_WRITEMASK	4xB	GetBooleanv	True	Color write mask for draw buffer 0	4.2.2	color-buffer
COLOR_WRITEMASK	4xB	GetBooleanIndexedvEXT	True	Color write mask for draw buffer i where i is specified as <index>	4.2.2	color-buffer

Issues

1. Should the extension provide support for per draw buffer index masks as well as per draw buffer color masks?

RESOLVED: No. Color index rendering is not interesting enough to warrant extending the API in this direction.

2. Should the API for specifying separate color write masks be based on DrawBuffers() (specifying an array of write masks at once)?

RESOLVED: No. There are two ways to mimic the DrawBuffers() API. A function, ColorMasks(), could take an an element count and an array of four element boolean arrays as parameters. Each four element boolean array contains a set of red, green, blue, and alpha write masks for a specific color buffer. An alternative is a ColorMasks() function that takes an element count and four parallel boolean arrays with one array per color channel. Neither approach is particularly clean. A cleaner approach, taken by ColorMaskIndexedEXT(), is to specify a color mask for a single draw buffer where the draw buffer is specified as a parameter to the function.

3. How should ColorMask() affect the per color buffer write masks?

RESOLVED: ColorMask() should set all color buffer write masks to the same values. This is backwards compatible with the way ColorMask() behaves in the absence of this extension.

4. What should GetBooleanv return when COLOR_WRITEMASK is queried?

RESOLVED: COLOR_WRITEMASK should return DRAW_BUFFER0_COLOR_WRITEMASK_EXT. This is backwards compatible with the way the query works without this extension. To query the writemask associated with a particular draw buffer, an application can use GetBooleanIndexedvEXT.

5. How are separate blend enables controlled? Should a new function be introduced, or do Enable() and Disable() provide sufficient functionality?

RESOLVED: This extension introduces new functions EnableIndexedEXT and DisableIndexedEXT that can be used to enable/disable individual states of a state array. These functions are introduced because there is a trend towards introducing arrays of state. Rather than creating enums for each index in the array, it is better to give applications a mechanism for accessing a particular element of the state array given the name of the state and an index into the array.

6. What effect does enabling or disabling blending using BLEND have on per draw buffer blend enables?

RESOLVED: BLEND, used with Enable() and Disable(), should enable or disable all per draw buffer blend enables. This is similar to the way that ColorMask() affects the per draw buffer write masks.

Revision History

None

Name

    EXT_draw_instanced

Name Strings

    GL_EXT_draw_instanced

Contact

Michael Gold, NVIDIA Corporation (gold 'at' nvidia.com)

Status

    Shipping for GeForce 8 Series (November 2006)

Version

    Last Modified Date: November 6, 2006
    Author Revision: 1.4

Number

    327

Dependencies

    OpenGL 2.0 is required.

    EXT_gpu_shader4 or NV_vertex_shader4 is required.

Overview

    This extension provides the means to render multiple instances of
    an object with a single draw call, and an "instance ID" variable
    which can be used by the vertex program to compute per-instance
    values, typically an object's transform.

New Tokens

    None

New Procedures and Functions

```
void DrawArraysInstancedEXT(enum mode, int first, sizei count,
        sizei primcount);
void DrawElementsInstancedEXT(enum mode, sizei count, enum type,
        const void *indices, sizei primcount);
```

Additions to Chapter 2 of the OpenGL 2.0 Specification (OpenGL Operation)

Modify section 2.8 (Vertex Arrays), p. 23

(insert before the final paragraph, p. 30)

The internal counter <instanceID> is a 32-bit integer value which may be read by a vertex program as <vertex.instance>, as described in section 2.X.3.2, or vertex shader as <gl_InstanceID>, as described in section 2.15.4.2. The value of this counter is always zero, except as noted below.

The command

```
    void DrawArraysInstancedEXT(enum mode, int first, sizei count,
            sizei primcount);
```

behaves identically to DrawArrays except that <primcount> instances of the range of elements are executed and the value of <instanceID> advances for each iteration. It has the same effect as:

```
    if (mode or count is invalid)
        generate appropriate error
    else {
        for (i = 0; i < primcount; i++) {
            instanceID = i;
            DrawArrays(mode, first, count, i);
        }
        instanceID = 0;
    }
```

The command

```
    void DrawElementsInstancedEXT(enum mode, sizei count, enum type,
            const void *indices, sizei primcount);
```

behaves identically to DrawElements except that <primcount> instances of the set of elements are executed, and the value of <instanceID> advances for each iteration. It has the same effect as:

```
    if (mode, count, or type is invalid )
        generate appropriate error
    else {
        for (int i = 0; i < primcount; i++) {
            instanceID = i;
            DrawElements(mode, count, type, indices, i);
        }
        instanceID = 0;
    }
```

Additions to Chapter 5 of the OpenGL 2.0 Specification (Special Functions)

The error INVALID_OPERATION is generated if DrawArraysInstancedEXT or DrawElementsInstancedEXT is called during display list compilation.

Dependencies on NV_vertex_program4

If NV_vertex_program4 is not supported, all references to vertex.instance are deleted.

Dependencies on EXT_gpu_shader4

If EXT_gpu_shader4 is not supported, all references to gl_InstanceID are deleted.

Errors

INVALID_ENUM is generated by DrawElementsInstancedEXT if <type> is not one of UNSIGNED_BYTE, UNSIGNED_SHORT or UNSIGNED_INT.

INVALID_VALUE is generated by DrawArraysInstancedEXT if <first> is less than zero.

Issues

(1) Should instanceID be provided by this extension, or should it be provided by EXT_gpu_shader4, thus creating a dependence on that spec?

Resolved: While this extension could stand alone, its utility would be limited without the additional functionality provided by EXT_gpu_shader4; also, the spec language is cleaner if EXT_gpu_shader4 assumes instanceID is always available, even if its value is always zero without this extension.

(2) Should MultiDrawArrays and MultiDrawElements affect the value of instanceID?

Resolved: No, this may cause implementation difficulties and is considered unlikely to provide any real benefit.

(3) Should DrawArraysInstanced and DrawElementsInstanced be compiled into display lists?

Resolved: No, calling these during display list compilation generate INVALID_OPERATION.

Revision History

None

Name

    EXT_draw_range_elements

Name Strings

    GL_EXT_draw_range_elements

Version

    $Date: 1997/5/19

Number

    112

Status

    Superceded by OpenGL 1.2 functionaltity.
    See section 2.8 (page 25) of the OpenGL 1.2.1 specification.

Proposal

Add a new vertex array rendering command:

```
void glDrawRangeElementsEXT(
        GLenum mode,
        GLuint start,
        GLuint end,
        GLsizei count,
        GLenum type,
        const GLvoid *indices
);
```

Add two implementation-dependent limits for describing data size recommendations for glDrawRangeElementsEXT:

```
GL_MAX_ELEMENTS_VERTICES_EXT  0x80E8
GL_MAX_ELEMENTS_INDICES_EXT   0x80E9
```

glDrawRangeElementsEXT is a restricted form of glDrawElements. All vertices referenced by indices must lie between start and end inclusive. Not all vertices between start and end must be referenced, however unreferenced vertices may be sent through some of the vertex pipeline before being discarded, reducing performance from what could be achieved by an optimal index set. Index values which lie outside the range will cause implementation-dependent results.

glDrawRangeElementsEXT may also be further constrained to only operate at maximum performance for limited amounts of data. Implementations may advertise recommended maximum amounts of vertex and index data using the GL_MAX_ELEMENTS_VERTICES_EXT and GL_MAX_ELEMENTS_INDICES_EXT enumerants. If a particular call to glDrawRangeElementsEXT has (end-start+1) greater than GL_MAX_ELEMENTS_VERTICES_EXT or if count is greater than GL_MAX_ELEMENTS_INDICES_EXT then the implementation may be forced to process the data less efficiently than it could have with less data. An implementation which has no effective limits can advertise the maximum integer value for the two enumerants. An implementation must always process a glDrawRangeElementsEXT call with valid parameters regardless of the amount of data passed in the call.

GL_INVALID_VALUE will be returned if end is less than start. Other errors are as for glDrawElements.

Motivation:
Rendering primitives from indexed vertex lists is a fairly common graphics operation, particularly in modeling applications such as VRML viewers. OpenGL 1.1 added support for the glDrawElements API to allow rendering of primitives by indexing vertex array data.

The specification of glDrawElements does not allow optimal performance for some OpenGL implementations, however. In particular, it has no restrictions on the number of indices given, the number of unique vertices referenced nor a direct indication of the set of unique vertices referenced by the given indices. This forces some OpenGL implementations to walk the index data given, building up a separate list of unique vertex references for later use in the pipeline. Additionally, since some OpenGL implementations have internal limitations on how many vertices they can deal with simultaneously the unbounded nature of glDrawElements requires the implementation to be prepared to segment the input data and do multiple passes. These preprocessing steps can consume a significant amount of time.

Such preprocessing can be done once and stored when building display lists but this only works for objects whose geometry does not change. Applications using morphing objects or other objects that are changing dynamically cannot take advantage of display lists and so must pay the preprocessing penalty on every redraw.

glDrawRangeElementsEXT is designed to avoid the preprocessing steps which may be necessary for glDrawElements. As such it does not have the flexibility of glDrawElements but it is sufficiently functional for a large class of applications to benefit from its use.
glDrawRangeElementsEXT enhances glDrawElements in two ways:
1. The set of unique vertices referenced by the indices is explicitly indicated via the start and end parameters, removing the necessity to determine this through examination of the index data. The implementation is given a contiguous chunk of vertex data that it can immediately begin streaming through the vertex pipeline.
2. Recommended limits on the amount of data to be processed can be indicated by the implementation through GL_MAX_ELEMENTS_VERTICES_EXT and GL_MAX_ELEMENTS_INDICES_EXT. If an application respects these limits it removes the need to split the incoming data into multiple chunks since the maximums can be set to the optimal values for the implementation to handle in one pass.

The first restriction isn't particularly onerous for applications since they can always call glDrawElements in the case where they cannot or do not know whether they can call glDrawRangeElementsEXT. Performance should be at least as good as it was calling glDrawElements alone. The second point isn't really a restriction as glDrawRangeElementsEXT doesn't fail if the data size limits are exceeded.

OpenGL implementation effort is also minimal. For implementations where glDrawElements performance is not affected by preprocessing glDrawRangeElementsEXT can be implemented simply as a call to glDrawElements and the maximums set to the maximum integer value. For the case where glDrawElements is doing non-trivial preprocessing there is probably already an underlying routine that takes consecutive, nicely sectioned index and vertex chunks that glDrawRangeElementsEXT can plug directly in to.

Design Decisions

The idea of providing a set of vertex indices along with a set of element indices was considered but dropped as it still may require some preprocessing, although there is some reduction in overhead from glDrawElements. The implementation may require internal vertex data to be contiguous, in which case a gather operation would have to be performed with the vertex index list before vertex data could be processed. It is expected that most apps will keep vertex data for particular elements packed consecutively anyway so the added flexibility of a vertex index list would potentially impose overhead with little expected benefit. In the case where a vertex index list really is necessary to avoid performance penalties due to sparse vertex usage glDrawElements should provide performance similar to what such an API would have.

The restriction on maximum data size cannot easily be lifted without potential performance implications. For implementations which have an internal maximum vertex buffer size it would be necessary to break up large data sets into multiple chunks. Splitting indexed data requires walking the indices and gathering those that fall within particular chunks into sets for processing, a time-consuming operation. Splitting the indices themselves is easier but still requires some processing to handle connected primitives that cross a split.

Name

    EXT_framebuffer_blit

Name Strings

    GL_EXT_framebuffer_blit

Contributors

    Michael Gold
    Evan Hart
    Jeff Juliano
    Jon Leech
    Bill Licea-Kane
    Barthold Lichtenbelt
    Brian Paul
    Ian Romanick
    John Rosasco
    Jeremy Sandmel
    Eskil Steenberg

Contact

    Michael Gold, NVIDIA Corporation (gold 'at' nvidia.com)

Status

    Complete. Approved by the ARB "superbuffers" working group on
    November 8, 2005.

Version

    Last Modified Date: September 29, 2006
    Author Revision: 14

Number

    316

Dependencies

    OpenGL 1.1 is required.

    EXT_framebuffer_object is required.

    The extension is written against the OpenGL 1.5 specification.

    ARB_color_buffer_float affects the definition of this extension.

Overview

    This extension modifies EXT_framebuffer_object by splitting the
    framebuffer object binding point into separate DRAW and READ
    bindings. This allows copying directly from one framebuffer to
    another. In addition, a new high performance blit function is

    added to facilitate these blits and perform some data conversion
    where allowed.

IP Status

    No known IP claims.

New Procedures and Functions

```
    void BlitFramebufferEXT(int srcX0, int srcY0, int srcX1, int srcY1,
                            int dstX0, int dstY0, int dstX1, int dstY1,
                            bitfield mask, enum filter);
```

New Tokens

    Accepted by the <target> parameter of BindFramebufferEXT,
    CheckFramebufferStatusEXT, FramebufferTexture{1D|2D|3D}EXT,
    FramebufferRenderbufferEXT, and
    GetFramebufferAttachmentParameterivEXT:

```
    READ_FRAMEBUFFER_EXT                0x8CA8
    DRAW_FRAMEBUFFER_EXT                0x8CA9
```

    Accepted by the <pname> parameters of GetIntegerv, GetFloatv, and
    GetDoublev:

```
    DRAW_FRAMEBUFFER_BINDING_EXT        0x8CA6 // alias FRAMEBUFFER_BINDING_EXT
    READ_FRAMEBUFFER_BINDING_EXT        0x8CAA
```

Additions to Chapter 2 of the OpenGL 1.5 Specification (OpenGL Operation)

Append the following to section 2.6.1:

    "Calling Begin will result in an INVALID_FRAMEBUFFER_OPERATION_EXT
    error if the object bound to DRAW_FRAMEBUFFER_BINDING_EXT is not
    "framebuffer complete" (section 4.4.4.2)."

Additions to Chapter 3 of the OpenGL 1.5 Specification (Rasterization)

Add to section 3.6.3, at the end of the subsection titled "Alternate Color Table Specification Commands":

    "Calling CopyColorTable or CopyColorSubTable will result in an
    INVALID_FRAMEBUFFER_OPERATION_EXT error if the object bound to
    READ_FRAMEBUFFER_BINDING_EXT is not "framebuffer complete"
    (section 4.4.4.2)."

Add to section 3.6.3, at the end of the subsection titled "Alternate Convolution Filter Specification Commands":

    "Calling CopyConvolutionFilter1D or CopyConvolutionFilter2D will
    result in an INVALID_FRAMEBUFFER_OPERATION_EXT error if the object
    bound to READ_FRAMEBUFFER_BINDING_EXT is not "framebuffer
    complete" (section 4.4.4.2)."

In section 3.6.4, modify the final paragraph of the definition of DrawPixels as follows:

"Calling DrawPixels will result in an INVALID_FRAMEBUFFER_OPERATION_EXT error if the object bound to DRAW_FRAMEBUFFER_BINDING_EXT is not "framebuffer complete" (section 4.4.4.2)."

Add the following to section 3.7, following the description of Bitmap:

"Calling Bitmap will result in an INVALID_FRAMEBUFFER_OPERATION_EXT error if the object bound to DRAW_FRAMEBUFFER_BINDING_EXT is not "framebuffer complete" (section 4.4.4.2)."

Append the following to section 3.8.2:

"Calling CopyTexImage3D, CopyTexSubImage3D, CopyTexImage2D, CopyTexSubImage2D, CopyTexImage1D or CopyTexSubImage1D will result in an INVALID_FRAMEBUFFER_OPERATION_EXT error if the object bound to READ_FRAMEBUFFER_BINDING_EXT is not "framebuffer complete" (section 4.4.4.2)."

Additions to Chapter 4 of the OpenGL 1.5 Specification (Per-Fragment Operations and the Frame Buffer)

Change the first word of Chapter 4 from "The" to "A".

Append to the introduction of Chapter 4:

"Conceptually, the GL has two active framebuffers; the draw framebuffer is the destination for rendering operations, and the read framebuffer is the source for readback operations. The same framebuffer may be used for both drawing and reading. Section 4.4.1 describes the mechanism for controlling framebuffer usage."

Modify the last paragraph of section 4.1.1 as follows:

"While an application-created framebuffer object is bound to DRAW_FRAMEBUFFER_EXT, the pixel ownership test always passes."

Modify the last sentence of the second to last paragraph of section 4.2.4 as follows:

"If there is no accumulation buffer, or if the DRAW_FRAMEBUFFER_EXT and READ_FRAMEBUFFER_EXT bindings (section 4.4.4.2) do not refer to the same object, or if the GL is in color index mode, Accum generates the error INVALID_OPERATION."

Add to 4.3.2 (Reading Pixels), right before the subsection titled "Obtaining Pixels from the Framebuffer":

"Calling ReadPixels generates INVALID_FRAMEBUFFER_OPERATION_EXT if the object bound to READ_FRAMEBUFFER_BINDING_EXT is not "framebuffer complete" (section 4.4.4.2)."

In section 4.3.2, modify the definition of ReadBuffer as follows:

"The command

```
     void ReadBuffer( enum src );
```

takes a symbolic constant as argument. <src> must be one of the values from tables 4.4 or 10.nnn. Otherwise, INVALID_ENUM is generated. Further, the acceptable values for <src> depend on whether the GL is using the default window-system-provided framebuffer (i.e., READ_FRAMEBUFFER_BINDING_EXT is zero), or an application-created framebuffer object (i.e., READ_FRAMEBUFFER_BINDING_EXT is non-zero). For more information about application-created framebuffer objects, see section 4.4.

When READ_FRAMEBUFFER_BINDING_EXT is zero, i.e. the default window-system-provided framebuffer, <src> must be one of the values listed in table 4.4. FRONT and LEFT refer to the front left buffer, BACK refers to the back left buffer, and RIGHT refers to the front right buffer. The other constants correspond directly to the buffers that they name. If the requested buffer is missing, then the error INVALID_OPERATION is generated. For the default window-system-provided framebuffer, the initial setting for ReadBuffer is FRONT if there is no back buffer and BACK otherwise.

ReadBuffer will set the read buffer for input colors other than 0 to NONE.

Modify the first sentence of section 4.3.3 as follows:

"CopyPixels transfers a rectangle of pixel values from one region of the read framebuffer to another in the draw framebuffer."

Add the following text to section 4.3.3, page 194, inside the definition of CopyPixels:

"Finally, the behavior of several GL operations is specified "as if the arguments were passed to CopyPixels." These operations include: CopyTex{Sub}Image*, CopyColor{Sub}Table, and CopyConvolutionFilter*. INVALID_FRAMEBUFFER_OPERATION_EXT will be generated if an attempt is made to execute one of these operations, or CopyPixels, while the object bound to READ_FRAMEBUFFER_BINDING_EXT is not "framebuffer complete" (as defined in section 4.4.4.2). Furthermore, an attempt to execute CopyPixels will generate INVALID_FRAMEBUFFER_OPERATION_EXT while the object bound to DRAW_FRAMEBUFFER_BINDING_EXT is not "framebuffer complete"."

Append to section 4.3.3:

"BlitFramebufferEXT transfers a rectangle of pixel values from one region of the read framebuffer to another in the draw framebuffer. There are some important distinctions from CopyPixels, as described below.

```
BlitFramebufferEXT(int srcX0, int srcY0, int srcX1, int srcY1,
                   int dstX0, int dstY0, int dstX1, int dstY1,
                   bitfield mask, enum filter);
```

<mask> is the bitwise OR of a number of values indicating which buffers are to be copied. The values are COLOR_BUFFER_BIT, DEPTH_BUFFER_BIT, and STENCIL_BUFFER_BIT, which are described in section 4.2.3. The pixels corresponding to these buffers are copied from the source rectangle, bound by the locations (srcX0, srcY0) and (srcX1, srcY1), to the destination rectangle, bound by the locations (dstX0, dstY0) and (dstX1, dstY1). The lower bounds of the rectangle are inclusive, while the upper bounds are exclusive.

If the source and destination rectangle dimensions do not match, the source image is stretched to fit the destination rectangle. <filter> must be LINEAR or NEAREST and specifies the method of interpolation to be applied if the image is stretched. LINEAR filtering is allowed only for the color buffer; if <mask> includes DEPTH_BUFFER_BIT or STENCIL_BUFFER_BIT, and filter is not NEAREST, no copy is performed and an INVALID_OPERATION error is generated. If the source and destination dimensions are identical, no filtering is applied. If either the source or destination rectangle specifies a negative dimension, the image is reversed in the corresponding direction. If both the source and destination rectangles specify a negative dimension for the same direction, no reversal is performed.

If the source and destination buffers are identical, and the source and destination rectangles overlap, the result of the blit operation is undefined.

The pixel copy bypasses the fragment pipeline. The only fragment operations which affect the blit are the pixel ownership test and the scissor test.

If a buffer is specified in <mask> and does not exist in both the read and draw framebuffers, the corresponding bit is silently ignored.

If the color formats of the read and draw framebuffers do not match, and <mask> includes COLOR_BUFFER_BIT, the pixel groups are converted to match the destination format as in CopyPixels, except that no pixel transfer operations apply and clamping behaves as if CLAMP_FRAGMENT_COLOR_ARB is set to FIXED_ONLY_ARB.

Calling CopyPixels or BlitFramebufferEXT will result in an INVALID_FRAMEBUFFER_OPERATION_EXT error if the objects bound to DRAW_FRAMEBUFFER_BINDING_EXT and READ_FRAMEBUFFER_BINDING_EXT are not "framebuffer complete" (section 4.4.4.2)."

Calling BlitFramebufferEXT will result in an INVALID_OPERATION error if <mask> includes DEPTH_BUFFER_BIT or STENCIL_BUFFER_BIT and the source and destination depth and stencil buffer formats do not match.

Modify the beginning of section 4.4.1 as follows:

"The default framebuffer for rendering and readback operations is provided by the windowing system. In addition, named framebuffer objects can be created and operated upon. The namespace for framebuffer objects is the unsigned integers, with zero reserved by the GL for the default framebuffer.

A framebuffer object is created by binding an unused name to DRAW_FRAMEBUFFER_EXT or READ_FRAMEBUFFER_EXT. The binding is effected by calling

```
    void BindFramebufferEXT(enum target, uint framebuffer);
```

with <target> set to the desired framebuffer target and <framebuffer> set to the unused name. The resulting framebuffer object is a new state vector, comprising all the state values listed in table 4.nnn, as well as one set of the state values listed in table 5.nnn for each attachment point of the framebuffer, set to the same initial values. There are MAX_COLOR_ATTACHMENTS_EXT color attachment points, plus one each for the depth and stencil attachment points.

BindFramebufferEXT may also be used to bind an existing framebuffer object to DRAW_FRAMEBUFFER_EXT or READ_FRAMEBUFFER_EXT. If the bind is successful no change is made to the state of the bound framebuffer object, and any previous binding to <target> is broken.

If a framebuffer object is bound to DRAW_FRAMEBUFFER_EXT or READ_FRAMEBUFFER_EXT, it becomes the target for rendering or readback operations, respectively, until it is deleted or another framebuffer is bound to the corresponding bind point. Calling BindFramebufferEXT with <target> set to FRAMEBUFFER_EXT binds the framebuffer to both DRAW_FRAMEBUFFER_EXT and READ_FRAMEBUFFER_EXT.

While a framebuffer object is bound, GL operations on the target to which it is bound affect the images attached to the bound framebuffer object, and queries of the target to which it is bound return state from the bound object. Queries of the values specified in table 6.31 (Implementation Dependent Pixel Depths) and table 8.nnn (Framebuffer-Dependent State Variables) are derived from the framebuffer object bound to DRAW_FRAMEBUFFER_EXT.

The initial state of DRAW_FRAMEBUFFER_EXT and READ_FRAMEBUFFER_EXT refers to the default framebuffer provided by the windowing system. In order that access to the default framebuffer is not lost, it is treated as a framebuffer object with the name of 0. The default framebuffer is therefore rendered to and read from while 0 is bound to the corresponding targets. On some implementations, the properties of the default framebuffer can change over time (e.g., in response to windowing system events such as attaching the context to a new windowing system drawable.)"

Change the description of DeleteFramebuffersEXT as follows:

"<framebuffers> contains <n> names of framebuffer objects to be deleted. After a framebuffer object is deleted, it has no attachments, and its name is again unused. If a framebuffer that is currently bound to one or more of the targets DRAW_FRAMEBUFFER_EXT or READ_FRAMEBUFFER_EXT is deleted, it is as though BindFramebufferEXT had been executed with the corresponding <target> and <framebuffer> zero. Unused names in <framebuffers> are silently ignored, as is the value zero."

In section 4.4.2.2, modify the first two sentences of the description of FramebufferRenderbufferEXT as follows:

"<target> must be DRAW_FRAMEBUFFER_EXT, READ_FRAMEBUFFER_EXT, or FRAMEBUFFER_EXT. If <target> is FRAMEBUFFER_EXT, it behaves as though DRAW_FRAMEBUFFER_EXT was specified. INVALID_OPERATION is generated if the value of the corresponding binding is zero."

In section 4.4.2.3, modify the first two sentences of the description of FramebufferTexturexDEXT as follows:

"In all three routines, <target> must be DRAW_FRAMEBUFFER_EXT, READ_FRAMEBUFFER_EXT, or FRAMEBUFFER_EXT. If <target> is FRAMEBUFFER_EXT, it behaves as though DRAW_FRAMEBUFFER_EXT was specified. INVALID_OPERATION is generated if the value of the corresponding binding is zero."

In section 4.4.4.2, modify the first sentence of the description of CheckFramebufferStatusEXT as follows:

"If <target> is not DRAW_FRAMEBUFFER_EXT, READ_FRAMEBUFFER_EXT or FRAMEBUFFER_EXT, INVALID_ENUM is generated. If <target> is FRAMEBUFFER_EXT, it behaves as though DRAW_FRAMEBUFFER_EXT was specified."

Modify section 4.4.4.3 as follows:

"Attempting to render to or read from a framebuffer which is not framebuffer complete will generate an INVALID_FRAMEBUFFER_OPERATION_EXT error."

Additions to Chapter 6 of the OpenGL 1.5 Specification (State and State Requests)

In section 6.1.3, modify the first sentence of the description of GetFramebufferAttachmentParameterivEXT as follows:

"<target> must be DRAW_FRAMEBUFFER_EXT, READ_FRAMEBUFFER_EXT or FRAMEBUFFER_EXT. If <target> is FRAMEBUFFER_EXT, it behaves as though DRAW_FRAMEBUFFER_EXT was specified."

GLX Protocol

```
    BlitFramebufferEXT
        2        44              rendering command length
        2        4330            rendering command opcode
        4        CARD32          source X0
        4        CARD32          source Y0
        4        CARD32          source X1
        4        CARD32          source Y1
        4        CARD32          destination X0
        4        CARD32          destination Y0
        4        CARD32          destination X1
        4        CARD32          destination Y1
        4        CARD32          mask
        4        ENUM            filter
```

Dependencies on ARB_color_buffer_float

The reference to CLAMP_FRAGMENT_COLOR_ARB in section 4.3.3 applies only if ARB_color_buffer_float is supported.

Errors

The error INVALID_FRAMEBUFFER_OPERATION_EXT is generated if BlitFramebufferEXT, DrawPixels, or CopyPixels is called while the draw framebuffer is not framebuffer complete.

The error INVALID_FRAMEBUFFER_OPERATION_EXT is generated if BlitFramebufferEXT, ReadPixels, CopyPixels, CopyTex{Sub}Image*, CopyColor{Sub}Table, or CopyConvolutionFilter* is called while the read framebuffer is not framebuffer complete.

The error INVALID_VALUE is generated by BlitFramebufferEXT if <mask> has any bits set other than those named by COLOR_BUFFER_BIT, DEPTH_BUFFER_BIT or STENCIL_BUFFER_BIT.

The error INVALID_OPERATION is generated if BlitFramebufferEXT is called and <mask> includes DEPTH_BUFFER_BIT or STENCIL_BUFFER_BIT and <filter> is not NEAREST.

The error INVALID_OPERATION is generated if BlitFramebufferEXT is called and <mask> includes DEPTH_BUFFER_BIT or STENCIL_BUFFER_BIT and the source and destination depth or stencil buffer formats do not match.

The error INVALID_ENUM is generated by BlitFramebufferEXT if <filter> is not LINEAR or NEAREST.

The error INVALID_OPERATION is generated if BlitFramebufferEXT is called within a Begin/End pair.

The error INVALID_ENUM is generated if BindFramebufferEXT, CheckFramebufferStatusEXT, FramebufferTexture{1D|2D|3D}EXT, FramebufferRenderbufferEXT, or GetFramebufferAttachmentParameterivEXT is called and <target> is not DRAW_FRAMEBUFFER_EXT, READ_FRAMEBUFFER_EXT or FRAMEBUFFER_EXT.

New State

    (modify table 3.nnn, "Framebuffer (state per framebuffer target binding
point)")

Get Value	Type	Get Command	Initial Ialue	Description	Section	Attribute
DRAW_FRAMEBUFFER_BINDING_EXT	Z+	GetIntegerv	0	framebuffer object bound to DRAW_FRAMEBUFFER_EXT	4.4.1	-
READ_FRAMEBUFFER_BINDING_EXT	Z+	GetIntegerv	0	framebuffer object to READ_FRAMEBUFFER_EXT	4.4.1	-

    Remove reference to FRAMEBUFFER_BINDING_EXT.

Sample Code

```
    /* Render to framebuffer object 2 */
    BindFramebufferEXT(DRAW_FRAMEBUFFER_EXT, 2);
    RenderScene();

    /* Blit contents of color buffer, depth buffer and stencil buffer
     * from framebuffer object 2 to framebuffer object 1.
     */
    BindFramebufferEXT(READ_FRAMEBUFFER_EXT, 2);
    BindFramebufferEXT(DRAW_FRAMEBUFFER_EXT, 1);
    BlitFramebufferEXT(0, 0, 640, 480,
                       0, 0, 640, 480,
                       GL_COLOR_BUFFER_BIT |
                       GL_DEPTH_BUFFER_BIT |
                       GL_STENCIL_BUFFER_BIT,
                       GL_NEAREST);

    /* Blit contents of color buffer from framebuffer object 1 to
     * framebuffer object 2, inverting the image in the X direction.
     */
    BindFramebufferEXT(READ_FRAMEBUFFER_EXT, 1);
    BindFramebufferEXT(DRAW_FRAMEBUFFER_EXT, 2);
    BlitFramebufferEXT(0, 0, 640, 480,
                       640, 0, 0, 480,
                       GL_COLOR_BUFFER_BIT,
                       GL_NEAREST);

    /* Blit color buffer from framebuffer object 1 to framebuffer
     * object 3 with a 2X zoom and linear filtering.
     */
    BindFramebufferEXT(READ_FRAMEBUFFER_EXT, 1);
    BindFramebufferEXT(DRAW_FRAMEBUFFER_EXT, 3);
    BlitFramebufferEXT(0, 0, 640, 480,
                       0, 0, 1280, 960,
                       GL_COLOR_BUFFER_BIT, GL_LINEAR);
```

Issues

```
    1) Should we pass in explicit source/dest rects instead of using
       the rasterpos/pixelzoom?

        Resolved: use explicit rects, so we don't need to perform
        multiple state changes.

    2) Should rects be (start,size) or (start,end)?

        Resolved: use (start,end).  This is a break from the past
        (scissor, viewport) but is more intuitive than allowing a
        negative size where mirrored zooms are desireable.

    3) What should we call the blit function?

        Resolved: BlitFramebufferEXT

    4) Should filtering apply to depth or stencil values?

        Resolved: No

    5) What happens if LINEAR is specified and DEPTH or STENCIL is in
       the mask?

        Resolved: Generate ERROR_INVALID_OPERATION

    6) What happens if READ_FRAMEBUFFER is NONE and a read is
       attempted?

        Resolved: Generate ERROR_INVALID_OPERATION

    7) Should we generalize binding point assignment with a single
       entry point and a parameter specifying read/write/whatever?

        Resolved: concensus leans toward separate Read/Draw entry
        points.

    8) Should we define READ_FRAMEBUFFER and DRAW_FRAMEBUFFER targets
       for BindFramebuffer instead of introducing a new level of
       indirection?

        Resolved: Yes.  Binding to the legacy target FRAMEBUFFER sets
        both DRAW_FRAMEBUFFER and READ_FRAMEBUFFER.  Querying
        FRAMEBUFFER_BINDING return the DRAW_FRAMEBUFFER_BINDING.

    9) What happens when a user queries framebuffer attributes,
       e.g. Get(RED_BITS)?  Is the result returned from
       READ_FRAMEBUFFER or DRAW_FRAMEBUFFER?  Do we need a new query?
       e.g.

       GetFramebufferParameteriv(int target, enum pname, int* value)

        Resolved: always return the value associated with the
        DRAW_FRAMEBUFFER.  Do not add a new query.
    10) How does Accum behave in the presence of separate READ/DRAW
        framebuffers?

        Resolved: Accum returns INVALID_OPERATION if the
        READ_FRAMEBUFFER and DRAW_FRAMEBUFFER bindings are not
        identical.

    11) Should blits be allowed between buffers of different bit sizes?

        Resolved: Yes, for color buffers only.  Attempting to blit
        between depth or stencil buffers of different size generates
        INVALID_OPERATION.

    12) Should we add support for multiple ReadBuffers, so that
        multiple color buffers may be copied with a single call to
        BlitFramebuffer?

        Resolved: No, we considered this but the behavior is awkward
        to define and the functionality is of limited use.

    13) How should BlitFramebuffer color space conversion be
        specified?  Do we allow context clamp state to affect the
        blit?

        Resolved: Blitting to a fixed point buffer always clamps,
        blitting to a floating point buffer never clamps.  The context
        state is ignored.

    14) Should overlapped blits be allowed?  Should they be guaranteed
        to work?

        Resolved: Overlapping blits are allowed but are undefined.
```

Revision History

```
    Revision 14, 2006/09/29
      - Changed the resolution of issue 12 to reflect the working
        group decision to abandon ReadBuffers.
      - Eliminated issues 15, 16 and 17 as they are no longer relevent.
      - Changed the resolution of issue 14 and the corresponding spec
        language to indicate that the result of an overlapping blit is
        undefined.
      - Changed the spec language to clarify that the lower bound of a
        blit rectangle is inclusive while the upper bound is
        exclusive.
      - Added a sample showing an inverted blit, to clarify the pixel
        addressing rules.
      - Clarified spec language and error behavior to indicate that
        blitting DEPTH and STENCIL buffers with LINEAR filtering is
        always disallowed, whether or not the blit is scaling.
    Revision 13, 2006/06/01 (Jeff Juliano)
      - Clarify errors generated when read and draw framebuffers are
        incomplete.
    Revision 12, 2005/12/22 (Jon Leech)
      - Assigned enumerant values. Add return type to BlitFramebufferEXT.
        Note INVALID_ENUM error if filter is not LINEAR or NEAREST.
    Revision 11, 2005/12/14
```

```
      - Added several missing conditions to the Errors section.
      - Changed status to "Complete".
    Revision 10, 2005/11/6
      - Removed all ReadBuffers discussion, as this functionality will
        be deferred.  Issues 15-17 are hereafter irrelevent.
    Revision 9, 2005/10/31
      - Resolved issue 16 and updated language to reflect this decision.
      - Minor language changes per feedback.
      - Added issue 17 and resolution, although language does not reflect this.
    Revision 8, 2005/10/20
      - Added ReadBuffersEXT language
      - Removed some redundant language in ReadBuffer
      - Re-opened issue 15 for further consideration
      - Added issue 16
    Revision 7, 2005/10/7
      - Added issues 13 and 14, and resolution for 11, 13, and 14.
      - Added dependency on ARB_color_buffer_float.
      - Removed multisample language, now covered in
        EXT_framebuffer_multisample.
      - Added framebuffer incomplete error language to spec proper.
      - Alias DRAW_FRAMEBUFFER_BINDING_EXT to FRAMEBUFFER_BINDING_EXT.
      - Updated Overview text to reflect the resolution to issue 8.
    Revision 6, 2005/9/26
      - Moved issues to the end, per new conventions.
      - Added new language referring to DRAW_FRAMEBUFFER and
        READ_FRAMEBUFFER bind points to sections 4.1.1, 4.4.1,
        4.4.2.2, 4.4.2.3, 4.4.4.2, 6.1.3 and Errors, and updated the
        example code, per resolution of issue 8.
      - Added language in section 4.4.1 specifying Get behavior, per
        resolution of issue 9.
      - Added language to section 4.2.4 describing new error behavior
        for Accum, per resolution of issue 10.
      - Added language to section 4.3.3 describing color format
        conversion, per resolution of issue 11.
    Revision 5, 2005/9/6
      - Added issues 8 - 11
      - Minor edits from reviewer feedback
    Revision 4, 2005/9/5
      - Added chapter 4 intro section
      - Added errors and state table information
      - Added sample code
      - fixed typos
    Revision 3, 2005/8/29
      - Converted to spec template
    Revision 2, 2005/7/18
      - Lots of new issues added and resolved
    Revision 1, 2005/7/5
      - Initial draft
```

Name

    EXT_framebuffer_multisample

Name Strings

    GL_EXT_framebuffer_multisample

Contributors

    Pat Brown
    Michael Gold
    Evan Hart
    Jeff Juliano
    Jon Leech
    Bill Licea-Kane
    Barthold Lichtenbelt
    Kent Lin
    Ian Romanick
    John Rosasco
    Jeremy Sandmel

Contacts

    Jeff Juliano, NVIDIA Corporation (jjuliano 'at' nvidia.com)
    Jeremy Sandmel, Apple Computer (jsandmel 'at' apple.com)

Status

    Complete
    Approved by the ARB "superbuffers" Working Group on November 8, 2005

Version

    Last Modified Date: November 6, 2006
    Revision: #6c

Number

    317

Dependencies

    Requires GL_EXT_framebuffer_object.

    Requires GL_EXT_framebuffer_blit.

    Written based on the wording of the OpenGL 1.5 specification.

Overview

    This extension extends the EXT_framebuffer_object framework to
    enable multisample rendering.

    The new operation RenderbufferStorageMultisampleEXT() allocates
    storage for a renderbuffer object that can be used as a multisample
    buffer. A multisample render buffer image differs from a

single-sample render buffer image in that a multisample image has a number of SAMPLES that is greater than zero. No method is provided for creating multisample texture images.

All of the framebuffer-attachable images attached to a framebuffer object must have the same number of SAMPLES or else the framebuffer object is not "framebuffer complete". If a framebuffer object with multisample attachments is "framebuffer complete", then the framebuffer object behaves as if SAMPLE_BUFFERS is one.

In traditional multisample rendering, where DRAW_FRAMEBUFFER_BINDING_EXT is zero and SAMPLE_BUFFERS is one, the GL spec states that "the color sample values are resolved to a single, displayable color each time a pixel is updated." There are, however, several modern hardware implementations that do not actually resolve for each sample update, but instead postpones the resolve operation to a later time and resolve a batch of sample updates at a time. This is OK as long as the implementation behaves "as if" it had resolved a sample-at-a-time. Unfortunately, however, honoring the "as if" rule can sometimes degrade performance.

In contrast, when DRAW_FRAMEBUFFER_BINDING_EXT is an application-created framebuffer object, MULTISAMPLE is enabled, and SAMPLE_BUFFERS is one, there is no implicit per-sample-update resolve. Instead, the application explicitly controls when the resolve operation is performed. The resolve operation is affected by calling BlitFramebufferEXT (provided by the EXT_framebuffer_blit extension) where the source is a multisample application-created framebuffer object and the destination is a single-sample framebuffer object (either application-created or window-system provided).

This design for multisample resolve more closely matches current hardware, but still permits implementations which choose to resolve a single sample at a time. If hardware that implementes the multisample resolution "one sample at a time" exposes EXT_framebuffer_multisample, it could perform the implicit resolve to a driver-managed hidden surface, then read from that surface when the application calls BlitFramebufferEXT.

Another motivation for granting the application explicit control over the multisample resolve operation has to do with the flexibility afforded by EXT_framebuffer_object. Previously, a drawable (window or pbuffer) had exclusive access to all of its buffers. There was no mechanism for sharing a buffer across multiple drawables. Under EXT_framebuffer_object, however, a mechanism exists for sharing a framebuffer-attachable image across several framebuffer objects, as well as sharing an image between a framebuffer object and a texture. If we had retained the "implicit" resolve from traditional multisampled rendering, and allowed the creation of "multisample" format renderbuffers, then this type of sharing would have lead to two problematic situations:

* Two contexts, which shared renderbuffers, might perform competing resolve operations into the same single-sample buffer with ambiguous results.

  * It would have introduced the unfortunate ability to use the
    single-sample buffer as a texture while MULTISAMPLE is ENABLED.

By using the BlitFramebufferEXT from EXT_framebuffer_blit as an explicit resolve to serialize access to the multisampeld contents and eliminate the implicit per-sample resolve operation, we avoid both of these problems.

Issues

Breaking from past convention, the issues section has been moved to the end of the document. It can be found after Examples, before Revision History.

New Procedures and Functions

```
void RenderbufferStorageMultisampleEXT(
        enum target, sizei samples,
        enum internalformat,
        sizei width, sizei height);
```

New Types

None.

New Tokens

Accepted by the <pname> parameter of GetRenderbufferParameterivEXT:

```
RENDERBUFFER_SAMPLES_EXT            0x8CAB
```

Returned by CheckFramebufferStatusEXT:

```
FRAMEBUFFER_INCOMPLETE_MULTISAMPLE_EXT  0x8D56
```

Accepted by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

```
MAX_SAMPLES_EXT                     0x8D57
```

Additions to Chapter 2 of the 1.5 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the OpenGL 1.5 Specification (Rasterization)

None

Additions to Chapter 4 of the OpenGL 1.5 Specification (Per-Fragment Operations and the Framebuffer)

Add to 4.3.2 (Reading Pixels), right before the subsection titled "Obtaining Pixels form the Framebuffer":

"ReadPixels generates INVALID_OPERATION if READ_FRAMEBUFFER_BINDING (section 4.4) is non-zero, the read framebuffer is framebuffer complete, and the value of SAMPLE_BUFFERS for the read framebuffer is greater than zero."

Modify the following text to section 4.3.3, page 194, that was added to the definition of CopyPixels by EXT_framebuffer_blit:

"Finally, the behavior of several GL operations is specified "as if the arguments were passed to CopyPixels." These operations include: CopyTex{Sub}Image*, CopyColor{Sub}Table, and CopyConvolutionFilter*. INVALID_FRAMEBUFFER_OPERATION_EXT will be generated if an attempt is made to execute one of these operations, or CopyPixels, while the object bound to READ_FRAMEBUFFER_BINDING_EXT (section 4.4) is not "framebuffer complete" (as defined in section 4.4.4.2). INVALID_OPERATION will be generated if the object bound to READ_FRAMEBUFFER_BINDING_EXT is "framebuffer complete" and the value of SAMPLE_BUFFERS is greater than zero.

Furthermore, an attempt to execute CopyPixels will generate INVALID_FRAMEBUFFER_OPERATION_EXT while the object bound to DRAW_FRAMEBUFFER_BINDING_EXT (section 4.4) is not "framebuffer complete".

In 4.3.3 (Copying Pixels), add to the section describing BlitFramebuffer that was added by EXT_framebuffer_blit.

"If SAMPLE_BUFFERS for the read framebuffer is greater than zero and SAMPLE_BUFFERS for the draw framebuffer is zero, the samples corresponding to each pixel location in the source are converted to a single sample before being written to the destination.

If SAMPLE_BUFFERS for the read framebuffer is zero and SAMPLE_BUFFERS for the draw framebuffer is greater than zero, the value of the source sample is replicated in each of the destination samples.

If SAMPLE_BUFFERS for both the read and draw framebuffers are greater than zero, and the value of SAMPLES for the read framebuffer matches the value of SAMPLES for the draw framebuffer, the samples are copied without modification from the read framebuffer to the draw framebuffer. Otherwise, no copy is performed and an INVALID_OPERATION error is generated.

Furthermore, if SAMPLE_BUFFERS for either the read framebuffer or draw framebuffer is greater than zero and the dimensions of the source and destination rectangles provided to BlitFramebuffer are not identical, no copy is performed and an INVALID_OPERATION error is generated."

Modification to 4.4.2.1 (Renderbuffer Objects)

Add, just above the definition of RenderbufferStorageEXT:

"The command

```
    void RenderbufferStorageMultisampleEXT(
        enum target, sizei samples,
        enum internalformat,
        sizei width, sizei height);
```

establishes the data storage, format, dimensions, and number of samples of a renderbuffer object's image. <target> must be RENDERBUFFER_EXT. <internalformat> must be RGB, RGBA, DEPTH_COMPONENT, STENCIL_INDEX, or one of the internal formats from table 3.16 or table 2.nnn that has a base internal format of RGB, RGBA, DEPTH_COMPONENT, or STENCIL_INDEX. <width> and <height> are the dimensions in pixels of the renderbuffer. If either <width> or <height> is greater than MAX_RENDERBUFFER_SIZE_EXT, or if <samples> is greater than MAX_SAMPLES_EXT, then the error INVALID_VALUE is generated. If the GL is unable to create a data store of the requested size, the error OUT_OF_MEMORY is generated.

Upon success, RenderbufferStorageMultisampleEXT deletes any existing data store for the renderbuffer image and the contents of the data store after calling RenderbufferStorageMultisampleEXT are undefined. RENDERBUFFER_WIDTH_EXT is set to <width>, RENDERBUFFER_HEIGHT_EXT is set to <height>, and RENDERBUFFER_INTERNAL_FORMAT_EXT is set to <internalformat>.

If <samples> is zero, then RENDERBUFFER_SAMPLES_EXT is set to zero. Otherwise <samples> represents a request for a desired minimum number of samples. Since different implementations may support different sample counts for multisampled rendering, the actual number of samples allocated for the renderbuffer image is implementation dependent. However, the resulting value for RENDERBUFFER_SAMPLES_EXT is guaranteed to be greater than or equal to <samples> and no more than the next larger sample count supported by the implementation.

Sized Internal Format	Base Internal format	S Bits
STENCIL_INDEX1_EXT	STENCIL_INDEX	1
STENCIL_INDEX4_EXT	STENCIL_INDEX	4
STENCIL_INDEX8_EXT	STENCIL_INDEX	8
STENCIL_INDEX16_EXT	STENCIL_INDEX	16

Table 2.nnn Desired component resolution for each sized internal format that can be used only with renderbuffers.

A GL implementation may vary its allocation of internal component resolution based on any RenderbufferStorage parameter (except target), but the allocation and chosen internal format must not be a function of any other state and cannot be changed once they are established."

```
    Modify the definiton of RenderbufferStorageEXT as follows:

    "The command

        void RenderbufferStorageEXT(enum target, enum internalformat,
                                    sizei width, sizei height);

     is equivalent to calling RenderbufferStorageMultisampleEXT with
     <samples> equal to zero."
```

Modification to 4.4.4.2 (Framebuffer Completeness)

```
    Add an entry to the bullet list:

    * The value of RENDERBUFFER_SAMPLES_EXT is the same for all attached
      images.
      { FRAMEBUFFER_INCOMPLETE_MULTISAMPLE_EXT }

    Also add a paragraph to the end of the section:

    "The values of SAMPLE_BUFFERS and SAMPLES are derived from the
    attachments of the currently bound framebuffer object.  If the
    current DRAW_FRAMEBUFFER_BINDING_EXT is not "framebuffer complete",
    then both SAMPLE_BUFFERS and SAMPLES are undefined.  Otherwise,
    SAMPLES is equal to the value of RENDERBUFFER_SAMPLES_EXT for the
    attached images (which all must have the same value for
    RENDERBUFFER_SAMPLES_EXT).  Further, SAMPLE_BUFFERS is one if
    SAMPLES is non-zero.  Otherwise, SAMPLE_BUFFERS is zero.
```

Additions to Chapter 5 of the OpenGL 1.5 Specification (Special Functions)

```
    Added to section 5.4, as part of the discussion of which commands
    are not compiled into display lists:

    "Certain commands, when called while compiling a display list, are
    not compiled into the display list but are executed immediately.
    These are: ..., RenderbufferStorageMultisampleEXT..."
```

Additions to Chapter 6 of the OpenGL 1.5 Specification (State and State Requests)

Modification to 6.1.3 (Enumerated Queries):

```
    In the list of state query functions, modify the definition of
    GetRenderbufferParameterivEXT as follows:

    "void GetRenderbufferParameterivEXT(enum target, enum pname,
                                        int* params);

        <target> must be RENDERBUFFER_EXT.  <pname> must be one of the
        symbolic values in table 8.nnn.

        If the renderbuffer currently bound to <target> is zero, then
        INVALID_OPERATION is generated.

        Upon successful return from GetRenderbufferParameterivEXT, if
        <pname> is RENDERBUFFER_WIDTH_EXT, RENDERBUFFER_HEIGHT_EXT,
        RENDERBUFFER_INTERNAL_FORMAT_EXT, or RENDERBUFFER_SAMPLES_EXT,
        then <params> will contain the width in pixels, height in
        pixels, internal format, or number of samples, respectively, of
        the renderbuffer currently bound to <target>.

        Otherwise, INVALID_ENUM is generated."
```

GLX Protocol

RenderbufferStorageMultisampleEXT

```
        2       24              rendering command length
        2       4331            rendering command opcode
        4       ENUM            target
        4       CARD32          samples
        4       ENUM            internalformat
        4       CARD32          width
        4       CARD32          height
```

Dependencies on EXT_framebuffer_object

EXT_framebuffer_object is required.

Dependencies on EXT_framebuffer_blit

EXT_framebuffer_blit is required. Technically, EXT_framebuffer_blit would not be required to support multisampled rendering, except for the fact that it provides the only method of doing a multisample resovle from a multisample renderbuffer.

Errors

The error INVALID_OPERATION_EXT is generated if ReadPixels, CopyPixels, CopyTex{Sub}Image*, CopyColor{Sub}Table, or CopyConvolutionFilter* is called while READ_FRAMEBUFFER_BINDING_EXT is non-zero, the read framebuffer is framebuffer complete, and the value of SAMPLE_BUFFERS for the read framebuffer is greater than zero.

The error OUT_OF_MEMORY is generated when RenderbufferStorageMultisampleEXT cannot storage of the specified size.

If both the draw and read framebuffers are framebuffer complete and both have a value of SAMPLE_BUFFERS that is greater than zero, then the error INVALID_OPERATION is generated if BlitFramebufferEXT is called and the values of SAMPLES for the draw and read framebuffers do not match.

If either the draw or read framebuffer is framebuffer complete and has a value of SAMPLE_BUFFERS that is greater than zero, then the error INVALID_OPERATION is generated if BlitFramebufferEXT is called and the specified source and destination dimensions are not identical.

    If RenderbufferStorageMultisampleEXT is called with a value of
    <samples> that is greater than MAX_SAMPLES_EXT, then the error
    INVALID_VALUE is generated.

New State

    (add to table 8.nnn, "Renderbuffers (state per renderbuffer object)")

Get Value	Type	Get Command	Initial Value	Description	Section	Attribute
RENDERBUFFER_SAMPLES_EXT	Z+	GetRenderbufferParameterivEXT	0	number of samples	4.4.2.1	-

To the table added by EXT_framebuffer_object called "Framebuffer
Dependent Values", table 9.nnn, add the following new framebuffer
dependent state.

Get Value	Type	Get Command	Minimum Value	Description	Section	Attribute
MAX_SAMPLES_EXT	Z+	GetIntegerv	1	Maximum number of samples supported for multisampling	4.4.2.1	-

Usage Examples

    XXX add examples XXX

Issues

    *(1) Should this be a separate extension or should it be included in
          a revision of EXT_framebuffer_object?*

          RESOLVED, separate extension

             Resolved by consensus, May 9, 2005

          This extension requires EXT_framebuffer_object but the reverse
          is not true. In addition, the cross framebuffer copy operation
          that will be used to handle the multisample resolution
          operation may be generally useful for non-multisampled
          rendering, but is pretty much required for multisampled
          rendering to be useful. Since we don't want
          EXT_framebuffer_object to require that functionality either, we
          split EXT_framebuffer_multisample into its own extension.
          EXT_framebuffer_multisample might include the "cross
          framebuffer copy" operation or might simply require the
          presence of that third extension. See issue (8).

    (2) What happens when <samples> is zero or one?

          RESOLVED, 0 = single sample, 1 = minimum multisample

             Resolved by consensus, May 9, 2005

          Zero means single sample, as if RenderbufferStorageEXT had been
          called instead of RenderbufferStorageMultisampleEXT. One means
          minimum number of samples supported by implementation.

There was a question if one should mean the same thing as single-sample (one sample), or if it should mean the minimum supported number of samples for multisample rendering. The rules for rasterizing in "multisample" mode are different than "non-multisample" mode. In the end, we decided that some implementations may wish to support a "one-sample" multisample buffer to allow for multipass multisampling where the sample location can be varied either by the implementation or perhaps explicitly by a "multisample location" extension.

(3) Is ReadPixels (or CopyPixels or CopyTexImage) permitted when bound to a multisample framebuffer object?

RESOLVED, no

Resolved by consensus, prior to May 9, 2005

No, those operations will produce INVALID_OPERATION. To read the contents of a multisample framebuffer, it must first be "downsampled" into a non-multisample destination, then read from there. For downsample, see EXT_framebuffer_blit.

The concern is fallback due to out of memory conditions. Even if no memory is available to allocate a temporary buffer at the time ReadPixels is called, an implementation should be able to make this work by pre-allocating a small tile and doing the downsample in tiles, or by falling back to software to copy a pixel at a time.

(4) Does the resolution from <samples> to RENDERBUFFER_SAMPLES_EXT depend on any other parameters to RenderbufferStorageMultisampleEXT, or must a given value of <samples> always resolve to the same number of actual samples?

RESOLVED, no, further, user must get at least what they asked for, or Storage call fails:

Resolved by consensus, May 23, 2005

Given the routine,

```
void RenderbufferStorageMultisampleEXT(
        enum target, uint samples,
        enum internalformat,
        uint width, uint height);
```

If an implementation supports several sample counts (say, 2x, 4x, 8x multisample), and the user requests a sample count of <samples>, the implementation must do one of the following:

- succeed in giving the user exactly <samples>, or

- succeed in giving the user a number of samples greater than <samples> but no more than the next highest number of samples supported by the implementation, or

- fail the request to RenderbufferStorageMultisampleEXT with an OUT_OF_MEMORY error

(5) *Is an implementation allowed to create single-sample storage when RenderbufferStorageMultisampleEXT is called with <samples> larger than one?*

RESOLVED, no

Resolved by consensus, May 23, 2005

No, by resolution of issue (4) above, the user must get at least what they asked for or higher, which precludes getting a single sampled format if they asked for a multisampled format.

(6) *Should OUT_OF_MEMORY be generated when RenderbufferStorageMultisampleEXT cannot create storage of the requested size?*

RESOLVED, yes

Resolved by consensus, May 23, 2005

Yes. Success or failure is determined by <width>, <height>, <internalformat>, and <samples>, and the implementation can always return OUT_OF_MEMORY. Note that while an implementation may give a different internal format with either higher or lower resolution per component than the internal requested, by issue of resolution (4), it must give at least the number of samples requested or it must fail the RenderbufferStorageMultisampleEXT call.

Update from June 2006 ARB meeting:

The appropriate error for the case where the number of samples is larger than the maximum supported by the implementation is INVALID_VALUE. To allow an application to know the maximum legal value, we add a GetInteger query MAX_SAMPLES.

(7) *Is there a query for the maximum size of <samples>?*

RESOLVED, no

Resolved by consensus, May 23, 2005

There was some discussion about whether it was useful to return a maximum sample count supported by the implementation as a convenenience to the developer so that the developer doesn't need to try increasingly smaller counts until it finds one that succeeds. However, in the end we decided that this was essentially the same problem already faced by the pixel format selection code in the glX/wgl/agl layer and so we decided not to add any special solution to this problem for multisampling with the framebuffer object API.

```
    (8)  Does this extension require our new framebuffer-to-framebuffer
         copy extension, EXT_framebuffer_blit, or is it merely affected
         by the presence of that extension.

         RESOLVED, EXT_framebuffer_blit is required.

         EXT_framebuffer_multisample by itself enable the user to
         perform multisampled rendering.  However, you can't copy or
         read from a multisampled renderbuffer using {Read|Copy}Pixels
         or CopyTex{Sub}Image - as per issue (3).  Consequently, there
         is no way to actually use the results of multisampled rendering
         without EXT_framebuffer_blit.  That makes the
         EXT_framebuffer_multisample extension arguably kind of useless
         without the EXT_framebuffer_blit.

         However, the reverse is not true.  The EXT_framebuffer_blit is
         useful on its own, which is why it is a separate extension from
         this one.

         So we decided to state that EXT_framebuffer_multisample
         requires EXT_framebuffer_blit instead of merely stating that
         that extension affects this one.

    (9)  Is DrawPixels allowed when the draw framebuffer is multisample?

         RESOLVED, yes

         This is no different than DrawPixels to a multisample window
         (framebuffer zero).  Note that ReadPixels and CopyPixels are
         disallowed when the read framebuffer is multisample.
```

Revision History

```
    #6c, November 6, 2006: jjuliano
        - changes from June #6 merged back in

    #6b, October 13, 2006: Jon Leech
        - added token values for MAX_SAMPLES_EXT and
          FRAMEBUFFER_INCOMPLETE_MULTISAMPLE_EXT.

    #6a, September 6, 2006: jsandmel
        - added language describing MAX_SAMPLES query
        - clarified that RenderbufferStorageMultisampleEXT can fail
          with INVALID_VALUE if <samples> is greater than MAX_SAMPLES

    #6, June 1, 2006: jjuliano
        - add missing errors to Errors section
        - clarify the modifications to 4.3.2 and 4.3.3.
        - add issue 9 to document that multisample DrawPixels is allowed

    #5, December 22, 2005: Jon Leech
        - added GLX protocol, assigned enumerant values
    #4, September 28, 2005: jsandmel, jjuliano
        - moved the multisample languge from GL_EXT_framebuffer_blit to
          this spec.
        - added description of using BlitFramebufferEXT for resolving
          multisample buffer
        - added language referring to DRAW_/READ_FRAMEBUFFER_BINDING
          instead of just FRAMEBUFFER_BINDING.
        - minor updates to reflect new EXT_framebuffer_blit spec
          that provides the multisample resolve function
        - resolve issue (8)
        - rename framebuffer_object_multisample to
          framebuffer_multisample

    #3, May 26, 2005: jsandmel
        - added recent workgroup resolutions
        - resolved issues (4), (5), (6), (7) based on decisions from the
          work group on May 9 and 23, 2005
        - added issue (8), does this extension require our new
          cross-framebuffer copy extension?
        - removed MAX_RENDERBUFFER_SAMPLES_EXT enum as per work group
          decision - issue (7)
        - changed prototype for RenderbufferStorageMultisampleEXT to use
          sizei for sample count

    #2, May 16, 2005: jsandmel
        - revised to account for recent work group meeting decisions
        - removed erroneous inclusion of GenerateMipmaps as a new
          function
        - resolved issue (1), this will be a separate extension
        - resolved issue (2), zero means non-multisample, one means
          minimum number of samples

    #1, May 9, 2005: jjuliano
        - first revision
```

Name

    EXT_framebuffer_object

Name Strings

    GL_EXT_framebuffer_object

Contributors

    Kurt Akeley
    Jason Allen
    Bob Beretta
    Pat Brown
    Matt Craighead
    Alex Eddy
    Cass Everitt
    Mark Galvan
    Michael Gold
    Evan Hart
    Jeff Juliano
    Mark Kilgard
    Dale Kirkland
    Jon Leech
    Bill Licea-Kane
    Barthold Lichtenbelt
    Kent Lin
    Rob Mace
    Teri Morrison
    Chris Niederauer
    Brian Paul
    Paul Puey
    Ian Romanick
    John Rosasco
    R. Jason Sams
    Jeremy Sandmel
    Mark Segal
    Avinash Seetharamaiah
    Folker Schamel
    Daniel Vogel
    Eric Werness
    Cliff Woolley

Contacts

    Jeff Juliano, NVIDIA Corporation (jjuliano 'at' nvidia.com)
    Jeremy Sandmel, Apple Computer (jsandmel 'at' apple.com)

Status

    Complete.
    Approved by the ARB "superbuffers" Working Group on January 31, 2005.
    Despite being controlled by the ARB WG, this is not an officially
    approved ARB extension at this time, thus the "EXT" tag.

Version

Last Modified Date: April 5, 2006
Revision: #118

Number

310

Dependencies

OpenGL 1.1 is required.

WGL_ARB_make_current_read affects the definition of this extension.

GLX 1.3 / GLX_SGI_make_current_read affects the definition of this extension.

ATI_draw_buffers affects the definition of this extension.

ARB_draw_buffers affects the definition of this extension.

ARB_fragment_program affects the definition of this extension.

ARB_fragment_shader affects the definition of this extension.

ARB_texture_rectangle affects the definition of this extension.

ARB_vertex_shader affects the definition of this extension.

EXT_packed_depth_stencil affects the definition of this extension.

NV_float_buffer affects the definition of this extension.

NV_texture_shader affects the definition of this extension.

Written based on the wording of the OpenGL 1.5 specification.

Overview

This extension defines a simple interface for drawing to rendering destinations other than the buffers provided to the GL by the window-system.

In this extension, these newly defined rendering destinations are known collectively as "framebuffer-attachable images". This extension provides a mechanism for attaching framebuffer-attachable images to the GL framebuffer as one of the standard GL logical buffers: color, depth, and stencil. (Attaching a framebuffer-attachable image to the accum logical buffer is left for a future extension to define). When a framebuffer-attachable image is attached to the framebuffer, it is used as the source and destination of fragment operations as described in Chapter 4.

By allowing the use of a framebuffer-attachable image as a rendering destination, this extension enables a form of "offscreen" rendering. Furthermore, "render to texture" is supported by allowing the images of a texture to be used as framebuffer-attachable images. A particular image of a texture object is selected for use as a framebuffer-attachable image by specifying the mipmap level, cube map face (for a cube map texture), and z-offset (for a 3D texture) that identifies the image. The "render to texture" semantics of this extension are similar to performing traditional rendering to the framebuffer, followed immediately by a call to CopyTexSubImage. However, by using this extension instead, an application can achieve the same effect, but with the advantage that the GL can usually eliminate the data copy that would have been incurred by calling CopyTexSubImage.

This extension also defines a new GL object type, called a "renderbuffer", which encapsulates a single 2D pixel image. The image of renderbuffer can be used as a framebuffer-attachable image for generalized offscreen rendering and it also provides a means to support rendering to GL logical buffer types which have no corresponding texture format (stencil, accum, etc). A renderbuffer is similar to a texture in that both renderbuffers and textures can be independently allocated and shared among multiple contexts. The framework defined by this extension is general enough that support for attaching images from GL objects other than textures and renderbuffers could be added by layered extensions.

To facilitate efficient switching between collections of framebuffer-attachable images, this extension introduces another new GL object, called a framebuffer object. A framebuffer object contains the state that defines the traditional GL framebuffer, including its set of images. Prior to this extension, it was the window-system which defined and managed this collection of images, traditionally by grouping them into a "drawable". The window-system API's would also provide a function (i.e., wglMakeCurrent, glXMakeCurrent, aglSetDrawable, etc.) to bind a drawable with a GL context (as is done in the WGL_ARB_pbuffer extension). In this extension however, this functionality is subsumed by the GL and the GL provides the function BindFramebufferEXT to bind a framebuffer object to the current context. Later, the context can bind back to the window-system-provided framebuffer in order to display rendered content.

Previous extensions that enabled rendering to a texture have been much more complicated. One example is the combination of ARB_pbuffer and ARB_render_texture, both of which are window-system extensions. This combination requires calling MakeCurrent, an operation that may be expensive, to switch between the window and the pbuffer drawables. An application must create one pbuffer per renderable texture in order to portably use ARB_render_texture. An application must maintain at least one GL context per texture format, because each context can only operate on a single pixelformat or FBConfig. All of these characteristics make ARB_render_texture both inefficient and cumbersome to use.

EXT_framebuffer_object, on the other hand, is both simpler to use and more efficient than ARB_render_texture. The EXT_framebuffer_object API is contained wholly within the GL API and has no (non-portable) window-system components. Under EXT_framebuffer_object, it is not necessary to create a second GL context when rendering to a texture image whose format differs from that of the window. Finally, unlike the pbuffers of ARB_render_texture, a single framebuffer object can facilitate rendering to an unlimited number of texture objects.

Glossary of Helpful Terms

logical buffer:
One of the color, depth, or stencil buffers of the framebuffer.

framebuffer:
The collection of logical buffers and associated state defining where the output of GL rendering is directed.

texture:
an object which consists of one or more 2D arrays of pixel images and associated state that can be used as a source of data during the texture-mapping process described in section 3.8.

texture image:
one of the 2D arrays of pixels that are part of a texture object as defined in section 3.8. Texture images contain and define the texels of the texture object.

renderbuffer:
A new type of storage object which contains a single 2D array of pixels and associated state that can be used as a destination for pixel data written during the rendering process described in Chapter 4.

renderbuffer image:
The 2D array of pixels that is part of a renderbuffer object. A renderbuffer image contains and defines the pixels of the renderbuffer object.

framebuffer-attachable image:
A 2D pixel image that can be attached to one of the logical buffer attachment points of a framebuffer object. Texture images and renderbuffer images are two examples of framebuffer-attachable images.

attachment point:
The set of state which references a specific framebuffer-attachable image, and allows that framebuffer-attachable image to be used to store the contents of a logical buffer of a framebuffer object. There is an attachment point state vector for each color, depth, and stencil buffer of a framebuffer.

attach:
The act of connecting one object to another object.

An "attach" operation is similar to a "bind" operation in that both represent a reference to the attached or bound object for the purpose of managing object lifetimes and both

```
            enable manipulation of the state of the attached or bound
            object.

            However, an "attach" is also different from a "bind" in that
            "binding" an unused object creates a new object, while
            "attaching" does not.  Additionally, "bind" establishes a
            connection between a context and an object, while "attach"
            establishes a connection between two objects.

            Finally, if object "A" is attached to object "B" and object
            "B" is bound to context "C", then in most respects, we treat
            "A" as if it is <implicitly> bound to "C".

        framebuffer attachment completeness:
            Similar to texture "mipmap" or "cube" completeness from
            section 3.8.10, defines a minimum set of criteria for
            framebuffer attachment points.  (for complete definition,
            see section 4.4.4.1)

        framebuffer completeness:
            Similar to texture "mipmap cube completeness", defines a
            composite set of "completeness" requirements and
            relationships among the attached framebuffer-attachable
            images.  (for complete definition, see section 4.4.4.2)
```

Issues

    Breaking from past convention, the very large issues section has
    been moved to the end of the document. It can be found after
    Examples, before Revision History.

New Procedures and Functions

```
    boolean IsRenderbufferEXT(uint renderbuffer);
    void BindRenderbufferEXT(enum target, uint renderbuffer);
    void DeleteRenderbuffersEXT(sizei n, const uint *renderbuffers);
    void GenRenderbuffersEXT(sizei n, uint *renderbuffers);

    void RenderbufferStorageEXT(enum target, enum internalformat,
                                sizei width, sizei height);

    void GetRenderbufferParameterivEXT(enum target, enum pname, int *params);

    boolean IsFramebufferEXT(uint framebuffer);
    void BindFramebufferEXT(enum target, uint framebuffer);
    void DeleteFramebuffersEXT(sizei n, const uint *framebuffers);
    void GenFramebuffersEXT(sizei n, uint *framebuffers);

    enum CheckFramebufferStatusEXT(enum target);
    void FramebufferTexture1DEXT(enum target, enum attachment,
                                 enum textarget, uint texture,
                                 int level);
    void FramebufferTexture2DEXT(enum target, enum attachment,
                                 enum textarget, uint texture,
                                 int level);
    void FramebufferTexture3DEXT(enum target, enum attachment,
                                 enum textarget, uint texture,
                                 int level, int zoffset);

    void FramebufferRenderbufferEXT(enum target, enum attachment,
                                    enum renderbuffertarget, uint renderbuffer);

    void GetFramebufferAttachmentParameterivEXT(enum target, enum attachment,
                                                enum pname, int *params);

    void GenerateMipmapEXT(enum target);
```

New Types

    None.

New Tokens

```
    Accepted by the <target> parameter of BindFramebufferEXT,
    CheckFramebufferStatusEXT, FramebufferTexture{1D|2D|3D}EXT,
    FramebufferRenderbufferEXT, and
    GetFramebufferAttachmentParameterivEXT:

        FRAMEBUFFER_EXT                     0x8D40

    Accepted by the <target> parameter of BindRenderbufferEXT,
    RenderbufferStorageEXT, and GetRenderbufferParameterivEXT, and
    returned by GetFramebufferAttachmentParameterivEXT:

        RENDERBUFFER_EXT                    0x8D41

    Accepted by the <internalformat> parameter of
    RenderbufferStorageEXT:

        STENCIL_INDEX1_EXT                  0x8D46
        STENCIL_INDEX4_EXT                  0x8D47
        STENCIL_INDEX8_EXT                  0x8D48
        STENCIL_INDEX16_EXT                 0x8D49

    Accepted by the <pname> parameter of GetRenderbufferParameterivEXT:

        RENDERBUFFER_WIDTH_EXT              0x8D42
        RENDERBUFFER_HEIGHT_EXT             0x8D43
        RENDERBUFFER_INTERNAL_FORMAT_EXT    0x8D44
        RENDERBUFFER_RED_SIZE_EXT           0x8D50
        RENDERBUFFER_GREEN_SIZE_EXT         0x8D51
        RENDERBUFFER_BLUE_SIZE_EXT          0x8D52
        RENDERBUFFER_ALPHA_SIZE_EXT         0x8D53
        RENDERBUFFER_DEPTH_SIZE_EXT         0x8D54
        RENDERBUFFER_STENCIL_SIZE_EXT       0x8D55
```

Accepted by the <pname> parameter of
GetFramebufferAttachmentParameterivEXT:

FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE_EXT	0x8CD0
FRAMEBUFFER_ATTACHMENT_OBJECT_NAME_EXT	0x8CD1
FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL_EXT	0x8CD2
FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE_EXT	0x8CD3
FRAMEBUFFER_ATTACHMENT_TEXTURE_3D_ZOFFSET_EXT	0x8CD4

Accepted by the <attachment> parameter of
FramebufferTexture{1D|2D|3D}EXT, FramebufferRenderbufferEXT, and
GetFramebufferAttachmentParameterivEXT

COLOR_ATTACHMENT0_EXT	0x8CE0
COLOR_ATTACHMENT1_EXT	0x8CE1
COLOR_ATTACHMENT2_EXT	0x8CE2
COLOR_ATTACHMENT3_EXT	0x8CE3
COLOR_ATTACHMENT4_EXT	0x8CE4
COLOR_ATTACHMENT5_EXT	0x8CE5
COLOR_ATTACHMENT6_EXT	0x8CE6
COLOR_ATTACHMENT7_EXT	0x8CE7
COLOR_ATTACHMENT8_EXT	0x8CE8
COLOR_ATTACHMENT9_EXT	0x8CE9
COLOR_ATTACHMENT10_EXT	0x8CEA
COLOR_ATTACHMENT11_EXT	0x8CEB
COLOR_ATTACHMENT12_EXT	0x8CEC
COLOR_ATTACHMENT13_EXT	0x8CED
COLOR_ATTACHMENT14_EXT	0x8CEE
COLOR_ATTACHMENT15_EXT	0x8CEF
DEPTH_ATTACHMENT_EXT	0x8D00
STENCIL_ATTACHMENT_EXT	0x8D20

Returned by CheckFramebufferStatusEXT():

FRAMEBUFFER_COMPLETE_EXT	0x8CD5
FRAMEBUFFER_INCOMPLETE_ATTACHMENT_EXT	0x8CD6
FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT_EXT	0x8CD7
FRAMEBUFFER_INCOMPLETE_DIMENSIONS_EXT	0x8CD9
FRAMEBUFFER_INCOMPLETE_FORMATS_EXT	0x8CDA
FRAMEBUFFER_INCOMPLETE_DRAW_BUFFER_EXT	0x8CDB
FRAMEBUFFER_INCOMPLETE_READ_BUFFER_EXT	0x8CDC
FRAMEBUFFER_UNSUPPORTED_EXT	0x8CDD

Accepted by GetIntegerv():

FRAMEBUFFER_BINDING_EXT	0x8CA6
RENDERBUFFER_BINDING_EXT	0x8CA7
MAX_COLOR_ATTACHMENTS_EXT	0x8CDF
MAX_RENDERBUFFER_SIZE_EXT	0x84E8

Returned by GetError():

INVALID_FRAMEBUFFER_OPERATION_EXT	0x0506

Additions to Chapter 2 of the 1.5 Specification (OpenGL Operation)

"The GL interacts with two classes of framebuffers: window-system-provided framebuffers and application-created framebuffers. There is always one window-system-provided framebuffer, while application-created framebuffers can be created as desired. These two types of framebuffer are distinguished primarily by the interface for configuring and managing their state.

The effects of GL commands on the window-system-provided framebuffer are ultimately controlled by the window-system that allocates framebuffer resources. It is the window-system that determines which portions of this framebuffer the GL may access at any given time and that communicates to the GL how those portions are structured. Therefore, there are no GL commands to configure the window-system-provided framebuffer. Similarly, display of framebuffer contents on a CRT monitor (including the transformation of individual framebuffer values by such techniques as gamma correction) is not addressed by the GL. Framebuffer configuration occurs outside of the GL in conjunction with the window-system.

The initialization of a GL context itself occurs when the window-system allocates a window for GL rendering and is influenced by the state of the window-system-provided framebuffer."

Additions to Chapter 3 of the OpenGL 1.5 Specification (Rasterization)

In section 3.6.4, page 102, add the following text to the definiton of DrawPixels:

"If the object bound to FRAMEBUFFER_BINDING_EXT is not "framebuffer complete" (as defined in section 4.4.4.2), then an attempt to call DrawPixels will generate the error INVALID_FRAMEBUFFER_OPERATION_EXT."

In section 3.8.8, add the following text immediately before the subsection "Mipmapping" on page 151:

"If all of the following conditions are satisfied, then the value of the selected Tau(ijk), Tau(ij), or Tau(i) in the above equations is undefined instead of referring to the value of the texel at location (i), (i,j), or (i,j,k). See Chapter 4 for discussion of framebuffer objects and their attachments.

* The current FRAMEBUFFER_BINDING_EXT names an application-created framebuffer object <F>.

* The texture is attached to one of the attachment points, <A>, of framebuffer object <F>.

```
  * TEXTURE_MIN_FILTER is NEAREST or LINEAR, and the value of
    FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL_EXT for attachment point
    <A> is equal to the value of TEXTURE_BASE_LEVEL

    -or-

    TEXTURE_MIN_FILTER is NEAREST_MIPMAP_NEAREST,
    NEAREST_MIPMAP_LINEAR, LINEAR_MIPMAP_NEAREST, or
    LINEAR_MIPMAP_LINEAR, and the value of
    FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL_EXT for attachment point
    <A> is within the the inclusive range from TEXTURE_BASE_LEVEL to
    q."
```

In subsection "Automatic Mipmap Generation" to section 3.8.8, replace the first paragraph with the following text:

"If the value of texture parameter GENERATE MIPMAP is TRUE and a change is made to the interior or border texels of the level[base] array of a mipmap by one of the texture image specification operations defined in sections 3.8.1 through 3.8.3, then a complete set of mipmap arrays (as defined in section 3.8.10) will be computed. Array levels level[base] + 1 through p are replaced with arrays derived from the modified level[base], regardless of their previous contents. All other mipmap arrays, including the level[base] array, are left unchanged by this computation."

Add a new subsection "Manual Mipmap Generation" to section 3.8.8, after "Automatic Mipmap Generation":

"Manual Mipmap Generation

Mipmaps can be generated manually with the command

```
  void GenerateMipmapEXT(enum target);
```

where <target> is one of TEXTURE_1D, TEXTURE_2D, TEXTURE_CUBE_MAP, or TEXTURE_3D. Mipmap generation affects the texture image attached to <target>. For cube map textures, INVALID_OPERATION is generated if the texture bound to <target> is not cube complete, as defined in section 3.8.10.

Mipmap generation replaces texture array levels level[base] + 1 through q with arrays derived from the level[base] array, as described above under Automatic Mipmap Generation. All other mipmap arrays, including the level[base] array, are left unchanged by this computation. For arrays in the range level[base] through q, inclusive, automatic and manual mipmap generation generate the same derived arrays, given identical level[base] arrays."

Modify the third paragraph of section 3.8.12, page 157, to read:

"Texture objects are deleted by calling

```
    void DeleteTextures( sizei n, uint *textures );
```

textures contains n names of texture objects to be deleted. After a texture object is deleted, it has no contents or dimensionality, and its name is again unused. If a texture that is currently bound to one of the targets TEXTURE 1D, TEXTURE 2D, TEXTURE 3D, or TEXTURE CUBE MAP is deleted, it is as though BindTexture had been executed with the same target and texture zero. Additionally, special care must be taken when deleting a texture if any of the images of the texture are attached to a framebuffer object. See section 4.4.2.3 for details.

Unused names in textures are silently ignored, as is the value zero."

Additions to Chapter 4 of the OpenGL 1.5 Specification (Per-Fragment Operations and the Framebuffer)

On page 170, in the introduction to chapter 4, modify the first three paragraphs to read as follows:

"The framebuffer consists of a set of pixels arranged as a two-dimensional array. The height and width of this array may vary from one GL implementation to another. For purposes of this discussion, each pixel in the framebuffer is simply a set of some number of bits. The number of bits per pixel may also vary depending on the particular GL implementation or context.

Further there are two classes of framebuffers: the default framebuffer supplied by the window-system-provided and application-created framebuffer objects. Every GL context has a single default window-system-provided framebuffer. Applications can optionally create additional non-displayable framebuffer objects. (For more information on application-created framebuffer objects see section 4.4)

Corresponding bits from each pixel in the framebuffer are grouped together into a bitplane; each bitplane contains a single bit from each pixel. These bitplanes are grouped into several logical buffers. These are the color, depth, stencil, and accumulation buffers. The color buffer actually consists of a number of buffers, and these color buffers serve related but slightly different purposes depending on whether the GL is bound to the default window-system-provided framebuffer or to an application-created framebuffer object.

For the default window-system-provided framebuffer, the color buffers are: the front left buffer, the front right buffer, the back left buffer, the back right buffer, and some number of auxiliary buffers. Typically, the contents of the front buffers are displayed on a color monitor while the contents of the back buffers are invisible. (Monoscopic contexts display only the front left buffer; stereoscopic contexts display both the front left and the front right buffers.) The contents of the auxiliary buffers are never visible. All color buffers must have the same number of bitplanes, although an implementation or context may choose not to provide right buffers, back buffers, or auxiliary buffers at all. Further, an implementation or context may not provide depth, stencil, or accumulation buffers.

For application-created framebuffer objects, the color buffers are not visible, and consequently the names of the color buffers are not related to a display device. The names of the color buffers of an application-created framebuffer object are: COLOR_ATTACHMENT0_EXT through COLOR_ATTACHMENTn_EXT. The names of the depth and stencil buffers are DEPTH_ATTACHMENT_EXT and STENCIL_ATTACHMENT_EXT. For more information about the buffers of an application-created framebuffer object, see section 4.4.2. To be considered framebuffer complete (see section 4.4.4), all color buffers attached to an application-created framebuffer object must have the same number of bitplanes. Depth and stencil buffers may optionally be attached to application-created framebuffers as well.

Color buffers consist of either unsigned integer color indices or R, G, B, and, optionally, A unsigned integer values. The number of bitplanes in each of the color buffers, the depth buffer, the stencil buffer, and the accumulation buffer is dependent on the currently bound framebuffer. For the default framebuffer, the number of bitplanes is fixed. For application-created framebuffer objects, however, the number of bitplanes in a given logical buffer may change if the state of the corresponding framebuffer attachment or attached image changes (see sections 4.4.2 and 4.4.5). If an accumulation buffer is provided, it must have at least as many bitplanes per R, G, and B color component as do the color buffers."

Add a new paragraph to the end of section 4.1.1, page 171:

"While an application-created framebuffer object is bound to FRAMEBUFFER_EXT, the pixel ownership test always passes. The pixels of application-created frambuffer objects are always owned by the GL, not the window system. Only while the window-system-provided framebuffer named zero is bound to FRAMEBUFFER_EXT does the window system control pixel ownership."

Change section 4.1.5, page 174, third paragraph, first two sentences to read as follows:

"<ref> is an integer reference value that is used in the unsigned stencil comparison. Stencil comparison operations and queries of <ref> use the value of <ref> clamped to the range [0, (2^s) - 1], where s is the current number of bits in the stencil buffer."

Replace the first three sentences of 4.1.10 "Logical Operation":

"Finally, a logical operation is applied between the incoming fragment's color or index values and the color or index values stored at the corresponding location in the framebuffer. The result replaces the values in the framebuffer at the fragment's (x[w], y[w]) coordinates. However, if the DRAW_BUFFERS state selects a single framebuffer-attachable image more than once, then an undefined value is written to those color buffers at the fragment's (x[w], y[x]) coordinates."

Change section 4.2.1, to read as follows:

"The first such operation is controlling the buffer into which color values are written. This is accomplished with

```
    void DrawBuffer( enum buf );
```

<buf> defines a buffer or set of buffers for writing. <buf> must be one of the values from tables 4.4 or 10.nnn. Otherwise, INVALID_ENUM is generated. In addition, acceptable values for <buf> depend on whether the GL is using the default window-system-provided framebuffer (i.e., FRAMEBUFFER_BINDING_EXT is zero), or an application-created framebuffer object (i.e., FRAMEBUFFER_BINDING_EXT is non-zero). In the initial state, the GL is bound to the the window-system-provided framebuffer. For more information about application-created framebuffer objects, see section 4.4.

If the GL is bound to the window-system-provided framebuffer, then <buf> must be one the values listed in table 4.4, which summarizes the constants and the buffers they indicate. In this case, <buf> is a symbolic constant specifying zero, one, two, or four buffers for writing. These constants refer to the four potentially visible buffers front left, front right, back left, and back right, and to the auxiliary buffers. Arguments other than AUXi that omit reference to LEFT or RIGHT refer to both left and right buffers. Arguments other than AUXi that omit reference to FRONT or BACK refer to both front and back buffers. AUXi enables drawing only to auxiliary buffer i. Each AUXi adheres to AUXi = AUX0 + i.

If the GL is bound to an application-created framebuffer object, <buf> must be one of the values listed in table 10.nnn, which summarizes the constants and the buffers they indicate. In this case, <buf> is a symbolic constant specifying a single color buffer for writing. Specifying COLOR_ATTACHMENTi_EXT enables drawing only to the image attached to the framebuffer at COLOR_ATTACHMENTi_EXT. Each COLOR_ATTACHMENTi_EXT adheres to COLOR_ATTACHMENTi_EXT = COLOR_ATTACHMENT0_EXT + i. The intial value of DRAW_BUFFER for application-created framebuffer objects is COLOR_ATTACHMENT0_EXT.

Symbolic Constant	Meaning
NONE	no buffer
COLOR_ATTACHMENT0	output fragment color to image attached at color attachment point 0
COLOR_ATTACHMENT1	output fragment color to image attached at color attachment point 1
...	...
COLOR_ATTACHMENTn	output fragment color to image attached at color attachment point n, where n is MAX_COLOR_ATTACHMENTS - 1

Table 10.nnn: Arguments to DrawBuffer(s) and ReadBuffer when the context is bound to an application-created framebuffer object, and the buffers they indicate

If the GL is bound to the window-system-provided framebuffer and DrawBuffer is supplied with a constant (other than NONE) that does not indicate any of the color buffers allocated to the GL context by the window-system (including those listed in table 10.nnn), then the error INVALID_OPERATION results.

If the GL is bound to the application-created framebuffer and DrawBuffer is supplied with a constant from table 4.4, or COLOR_ATTACHMENTm where m is greater than or equal to MAX_COLOR_ATTACHMENTS, then the error INVALID_OPERATION results.

If DrawBuffer is supplied with a constant that is neither legal for the window-system provided framebuffer nor legal for an application-created framebuffer object, then the error INVALID_ENUM results.

The command

```
    void DrawBuffers( sizei n, const enum *bufs );
```

defines the draw buffers to which all fragment colors are written. <n> specifies the number of buffers in <bufs>. <bufs> is a pointer to an array of symbolic constants specifying the buffer to which each fragment color is written.

Each enumerant listed in <bufs> must be one of the values from tables 10.nnn or 11.nnn. Otherwise, INVALID_ENUM is generated. Further, acceptable values for the constants in <bufs> depend on whether the GL is using the default window-system-provided framebuffer (i.e., FRAMEBUFFER_BINDING_EXT is zero), or an application-created framebuffer object (i.e., FRAMEBUFFER_BINDING_EXT is non-zero). For more information about application-created framebuffer objects, see section 4.4.

symbolic constant	front left	front right	back left	back right	aux i
NONE					
FRONT LEFT	X				
FRONT RIGHT		X			
BACK LEFT			X		
BACK RIGHT				X	
AUXi					X

Table 11.nnn: Arguments to DrawBuffers, when the context is bound to the window-system-provided framebuffer, and the buffers that they indicate.

If the GL is bound to the default window-system-provided framebuffer, then the each of the constants must be one of the values listed in table 11.nnn

If the GL is bound to an application-created framebuffer object, then each of the constants must be one of the values listed in table 10.nnn.

In both cases, the draw buffers being defined correspond in order to the respective fragment colors. The draw buffer for fragment colors beyond <n> is set to NONE.

The maximum number of draw buffers is implementation dependent and must be at least 1. The number of draw buffers supported can be queried by calling GetIntegerv with the symbolic constant MAX_DRAW_BUFFERS. INVALID_VALUE is generated if <n> is greater than MAX_DRAW_BUFFERS.

Except for NONE, a buffer may not appear more then once in the array pointed to by <bufs>. Specifying a buffer more then once will result in the error INVALID_OPERATION.

If fixed-function fragment shading is being performed, DrawBuffers specifies a set of draw buffers into which the fragment color is written.

If a fragment shader writes to "gl_FragColor", DrawBuffers specifies a set of draw buffers into which the single fragment color defined by "gl_FragColor" is written. If a fragment shader writes to gl FragData, DrawBuffers specifies a set of draw buffers into which each of the multiple fragment colors defined by "gl_FragData" are separately written. If a fragment shader writes to neither gl FragColor nor "gl_FragData", the values of the fragment colors following shader execution are undefined, and may differ for each fragment color.

For both window-system-provided and application-created framebuffers, the constants FRONT, BACK, LEFT, RIGHT, and FRONT_AND_BACK are not valid in the <bufs> array passed to DrawBuffers, and will result in the error INVALID_OPERATION. This restriction is because these constants may themselves refer to multiple buffers, as shown in table 4.4.

If the GL is bound to the window-system-provided framebuffer and DrawBuffers is supplied with a constant (other than NONE) that does not indicate any of the color buffers allocated to the GL context by the window-system, then the error INVALID_OPERATION results.

If the GL is bound to the application-created framebuffer and DrawBuffers is supplied with a constant from table 11.nnn, or COLOR_ATTACHMENTm where m is greater than or equal to MAX_COLOR_ATTACHMENTS, then the error INVALID_OPERATION results.

If DrawBuffers is supplied with a constant that is neither legal for the window-system provided framebuffer nor legal for an application-created framebuffer object, then the error INVALID_ENUM results.

Indicating a buffer or buffers using DrawBuffer or DrawBuffers causes subsequent pixel color value writes to affect the indicated buffers.

Specifying NONE as the draw buffer for a fragment color will inhibit that fragment color from being written to any buffer.

Monoscopic contexts include only left buffers, while stereoscopic contexts include both left and right buffers. Likewise, single buffered contexts include only front buffers, while double buffered contexts include both front and back buffers. The type of context is selected at GL initialization.

The state required to handle color buffer selection is an integer for each supported fragment color. For the default window-system-provided framebuffer, in the initial state, the draw buffer for fragment color zero is FRONT if there are no back buffers; otherwise it is BACK. For application-created framebuffer objects, the initial value of draw buffer for fragment color zero is COLOR_ATTACHMENT0_EXT. For both the window-system-provided framebuffer and application-created framebuffers, the initial state of draw buffers for fragment colors other then zero is NONE."

Modify section 4.2.2, page 185, third paragraph to read as follows:

"The command

```
        void StencilMask( uint mask );
```

controls the writing of particular bits into the stencil planes. The least significant s bits of mask comprise an integer mask (s is the number of bits in the stencil buffer), just as for IndexMask. The initial state is for the stencil plane mask to be 32 ones."

In section 4.3.2, page 190, modify the first two paragraphs of the definition of ReadBuffer to read as follows:

"The command

```
     void ReadBuffer( enum src );
```

takes a symbolic constant as argument. <src> must be one of the values from tables 4.4 or 10.nnn. Otherwise, INVALID_ENUM is generated. Further, the acceptable values for <src> depend on whether the GL is using the default window-system-provided framebuffer (i.e., FRAMEBUFFER_BINDING_EXT is zero), or an application-created framebuffer object (i.e., FRAMEBUFFER_BINDING_EXT is non-zero). For more information about application-created framebuffer objects, see section 4.4.

If the object bound to FRAMEBUFFER_BINDING_EXT is not "framebuffer complete" (as defined in section 4.4.4.2), then ReadPixels generates the error INVALID_FRAMEBUFFER_OPERATION_EXT. If ReadBuffer is supplied with a constant that is neither legal for the window-system provided framebuffer, nor legal for an application-created framebuffer object, then the error INVALID_ENUM results.

When FRAMEBUFFER_BINDING_EXT is zero, i.e. the default window-system-provided framebuffer, <src> must be one of the values listed in table 4.4. FRONT and LEFT refer to the front left buffer, BACK refers to the back left buffer, and RIGHT refers to the front right buffer. The other constants correspond directly to the buffers that they name. If the requested buffer is missing, then the error INVALID_OPERATION is generated. For the default window-system-provided framebuffer, the initial setting for ReadBuffer is FRONT if there is no back buffer and BACK otherwise.

When the GL is using an application-created framebuffer object, <src> must be one of the values listed in table 10.nnn, including NONE. In a manner analogous to how the DRAW_BUFFERs state is handled, specifying COLOR_ATTACHMENTi_EXT enables reading from the image attached to the framebuffer at COLOR_ATTACHMENTi_EXT. ReadPixels generates INVALID_OPERATION if it attempts to select a color buffer while READ_BUFFER is NONE. For application-created framebuffer objects, the initial setting for ReadBuffer is COLOR_ATTACHMENT0_EXT.

ReadPixels obtains values from the selected buffer from each pixel with lower left hand corner at (x+i, y+j) for (0 <= i < width) and (0 <= j < height); this pixel is said to be the ith pixel in the jth row. If any of these pixels lies outside of the window allocated to the current GL context, or outside of the image attached to the currently bound framebuffer object, then the values obtained for those pixels are undefined. When FRAMEBUFFER_BINDING_EXT is zero, results are also undefined for individual pixels that are not owned by the current context. Otherwise, ReadPixels obtains values from the selected buffer, regardless of how those values were placed there."

In section 4.3.2, "Reading Pixels", add a paragraph before "Conversion of RGBA values" on page 191:

"When FRAMEBUFFER_BINDING is non-zero, the red, green, blue, and alpha values are obtained by first reading the internal component values of the corresponding value in the image attached to the selected logical buffer. The internal component values are converted to red, green, blue, and alpha values as specified in the row of table 12.nnn corresponding to the internal format of the image attached to READ_BUFFER."

Add the following text to section 4.3.3, page 194, inside the definiton of CopyPixels:

"Furthermore, the behavior of several GL operations is specified "as if the arguments were passed to CopyPixels." These operations include: CopyTex{Sub}Image*, CopyColor{Sub}Table, and CopyConvolutionFilter*. INVALID_FRAMEBUFFER_OPERATION_EXT will be generated if an attempt is made to execute one of these operations, or CopyPixels, while the object bound to FRAMEBUFFER_BINDING_EXT is not "framebuffer complete" (as defined in section 4.4.4.2)."

Add a new section "Framebuffer Objects" after section 4.3:

"4.4 Framebuffer Objects

As described in chapters 1 and 2, GL renders into (and reads values from) a framebuffer. GL defines two classes of framebuffers: window-system-provided framebuffers and application-created framebuffers. For each GL context, there is a single framebuffer provided by the window-system, and there may also be one or more framebuffer objects created and managed by the application.

By default, the GL uses the window-system-provided framebuffer. The storage, dimensions, allocation, and format of the images attached to this framebuffer are managed entirely by the window-system. Consequently, the state of the window-system-provided framebuffer, including its images, can not be changed by the GL, nor can the window-system-provided framebuffer itself, or its images, be deleted by the GL.

The routines described in the following sections, however, can be used to create, destroy, and modify the state and attachments of application-created framebuffer objects.

Application-created framebuffer objects encapsulate the state of a framebuffer in a similar manner to the way texture objects encapsulate the state of a texture. In particular, a framebuffer object encapsulates state necessary to describe a collection of color, depth, stencil, accum, and aux logical buffers. For each logical buffer, a framebuffer-attachable image can be attached to the framebuffer to store the rendered output for that logical buffer. Examples of framebuffer-attachable images include texture images and renderbuffer images. Renderbuffers are described further in section 4.4.2.1

By allowing the images of a renderbuffer to be attached to a framebuffer, the GL provides a mechanism to support "off-screen" rendering. Further, by allowing the images of a texture to be attached to a framebuffer, the GL provides a mechanism to support "render to texture".

4.4.1 Binding and Managing Framebuffer Objects

The operations described in chapter 4 affect the images attached to the framebuffer object bound to the target FRAMEBUFFER_EXT. By default, framebuffer bound to the target FRAMEBUFFER_EXT is zero, specifying the default implementation dependent framebuffer provided by the windowing system. When the framebuffer bound to target FRAMEBUFFER_EXT is not zero, but instead names an application-created framebuffer object, then the operations described in chapter 4 affect the application-created framebuffer object rather than the default framebuffer.

The namespace for framebuffer objects is the unsigned integers, with zero reserved by the GL to refer to the default framebuffer. A framebuffer object is created by binding an unused name to the target FRAMEBUFFER_EXT. The binding is effected by calling

```
    void BindFramebufferEXT(enum target, uint framebuffer);
```

with <target> set to FRAMEBUFFER_EXT and <framebuffer> set to the unused name. The resulting framebuffer object is a new state vector, comprising all the state values listed in table 4.nnn, as well as one set of the state values listed in table 5.nnn for each attachment point of the framebuffer. There are MAX_COLOR_ATTACHMENTS_EXT color attachment points, plus one each for the depth and stencil attachment points.

BindFramebufferEXT may also be used to bind an existing framebuffer object to <target>. If the bind is successful no change is made to the state of the bound framebuffer object and any previous binding to <target> is broken. The current FRAMEBUFFER_EXT binding can be queried using GetIntegerv(FRAMEBUFFER_BINDING_EXT).

While a framebuffer object is bound to the target FRAMEBUFFER_EXT, GL operations on the target to which it is bound affect the images attached to the bound framebuffer object, and queries of the target to which it is bound return state from the bound object. In particular, queries of the values specified in table 6.31 (Implementation Dependent Pixel Depths) and table 8.nnn (Framebuffer-Dependent State Variables) are derived from the currently bound framebuffer object. The framebuffer object bound to the target FRAMEBUFFER_EXT is used as the destination of fragment operations and as the source of pixel reads such as ReadPixels, as described in chapter 4.

In the initial state, the reserved name zero is bound to the target FRAMEBUFFER_EXT. There is no application-created framebuffer object corresponding to the name zero. Instead, the name zero refers to the window-system-provided framebuffer. All queries and operations on the framebuffer while the name zero is bound to the target FRAMEBUFFER_EXT operate on this default framebuffer. On some implementations, the properties of the default window-system-provided framebuffer can change over time (e.g., in response to window-system events such as attaching the context to a new window-system drawable.)

Application-created framebuffer objects (i.e., those with a non-zero name) differ from the default window-system-provided framebuffer in a few important ways. First and foremost, unlike the window-system-provided framebuffer, application-created-framebuffers have modifiable attachment points for each logical buffer in the framebuffer. Framebuffer-attachable images can be attached to and detached from these attachment points, which are described further in section 4.4.2. Also, the size and format of the images attached to application-created framebuffers are controlled entirely within the GL interface, and are not affected by window-system events, such as pixel format selection, window resizes, and display mode changes.

Additionally, when rendering to or reading from an application created-framebuffer object,

- The pixel ownership test always succeeds. In other words, application-created framebuffer objects own all of their pixels.

- There are no visible color buffer bitplanes. This means there is no color buffer corresponding to the back, front, left, or right color bitplanes.

- The only color buffer bitplanes are the ones defined by the framebuffer attachment points named COLOR_ATTACHMENT0_EXT through COLOR_ATTACHMENTn_EXT.

- The only depth buffer bitplanes are the ones defined by the framebuffer attachment point DEPTH_ATTACHMENT_EXT.

- The only stencil buffer bitplanes are the ones defined by the framebuffer attachment point STENCIL_ATTACHMENT_EXT.

- There is no multisample buffer so the value of the implementation-dependent state variables SAMPLES and SAMPLE_BUFFERS are both 0

- There are no accum buffer bitplanes, so the value of the implementation-dependent state variables ACCUM_RED_BITS, ACCUM_GREEN_BITS, ACCUM_BLUE_BITS, and ACCUM_ALPHA_BITS, are all zero.

- There are no AUX buffer bitplanes, so the value of the implementation-dependent state variable AUX_BUFFERS is zero.

Framebuffer objects are deleted by calling

```
  void DeleteFramebuffersEXT(sizei n, uint *framebuffers);
```

<framebuffers> contains <n> names of framebuffer objects to be deleted. After a framebuffer object is deleted, it has no attachments, and its name is again unused. If a framebuffer that is currently bound to the target FRAMEBUFFER_EXT is deleted, it is as though BindFramebufferEXT had been executed with the <target> of FRAMEBUFFER_EXT and <framebuffer> of zero. Unused names in <framebuffers> are silently ignored, as is the value zero.

The command

```
  void GenFramebuffersEXT(sizei n, uint *ids);
```

returns <n> previously unused framebuffer object names in <ids>. These names are marked as used, for the purposes of GenFramebuffersEXT only, but they acquire state and type only when they are first bound, just as if they were unused.

4.4.2 Attaching Images to Framebuffer Objects

Framebuffer-attachable images may be attached to, and detached from, application-created framebuffer objects. In contrast, the image attachments of the window-system-provided framebuffer may not be changed by the GL.

A single framebuffer-attachable image may be attached to multiple application-created framebuffer objects, potentially avoiding some data copies, and possibly decreasing memory consumption.

For each logical buffer, the framebuffer object stores a set of state which defines the logical buffer's "attachment point". The "attachment point" state contains enough information to identify the single image attached to the attachment point, or to indicate that no image is attached. The per-logical buffer "attachment point" state is listed in table 5.nnn

There are two types of framebuffer-attachable images: the image of a renderbuffer object, and an image of a texture object.

4.4.2.1 Renderbuffer Objects

A renderbuffer is a data storage object containing a single image of a renderable internal format. GL provides the methods described below to allocate and delete a renderbuffer's image, and to attach a renderbuffer's image to a framebuffer object.

The name space for renderbuffer objects is the unsigned integers, with zero reserved for the GL. A renderbuffer object is created by binding an unused name to RENDERBUFFER_EXT. The binding is effected by calling

```
    void BindRenderbufferEXT( enum target, uint renderbuffer );
```

with <target> set to RENDERBUFFER_EXT and <renderbuffer> set to the unused name. If <renderbuffer> is not zero, then the resulting renderbuffer object is a new state vector, initialized with a zero-sized memory buffer, and comprising the state values listed in Table 8.nnn. Any previous binding to <target> is broken.

BindRenderbufferEXT may also be used to bind an existing renderbuffer object. If the bind is successful, no change is made to the state of the newly bound renderbuffer object, and any previous binding to <target> is broken.

While a renderbuffer object is bound, GL operations on the target to which it is bound affect the bound renderbuffer object, and queries of the target to which a renderbuffer object is bound return state from the bound object.

The name zero is reserved. A renderbuffer object cannot be created with the name zero. If <renderbuffer> is zero, then any previous binding to <target> is broken and the <target> binding is restored to the initial state.

In the initial state, the reserved name zero is bound to RENDERBUFFER_EXT. There is no renderbuffer object corresponding to the name zero, so client attempts to modify or query renderbuffer state for the target RENDERBUFFER_EXT while zero is bound will generate GL errors, as described in section 6.1.3.

Using GetIntegerv, the current RENDERBUFFER_EXT binding can be queried as RENDERBUFFER_BINDING_EXT.

Renderbuffer objects are deleted by calling

```
    void DeleteRenderbuffersEXT( sizei n, const uint *renderbuffers );
```

where <renderbuffers> contains n names of renderbuffer objects to be deleted. After a renderbuffer object is deleted, it has no contents, and its name is again unused. If a renderbuffer that is currently bound to RENDERBUFFER_EXT is deleted, it is as though BindRenderbufferEXT had been executed with the <target> RENDERBUFFER_EXT and <name> of zero. Additionally, special care must be taken when deleting a renderbuffer if the image of the renderbuffer is attached to a framebuffer object. (See section 4.4.2.2 for details). Unused names in <renderbuffers> are silently ignored, as is the value zero.

The command

```
    void GenRenderbuffersEXT( sizei n, uint *renderbuffers );
```

returns <n> previously unused renderbuffer object names in <renderbuffers>. These names are marked as used, for the purposes of GenRenderbuffersEXT only, but they acquire renderbuffer state only when they are first bound, just as if they were unused.

The command

```
    void RenderbufferStorageEXT(enum target, enum internalformat,
                                sizei width, sizei height);
```

establishes the data storage, format, and dimensions of a renderbuffer object's image. <target> must be RENDERBUFFER_EXT. <internalformat> must be color-renderable, depth-renderable, or stencil-renderable (as defined in section 4.4.4). <width> and <height> are the dimensions in pixels of the renderbuffer. If either <width> or <height> is greater than MAX_RENDERBUFFER_SIZE_EXT, the the error INVALID_VALUE is generated. If the GL is unable to create a data store of the requested size, the error OUT_OF_MEMORY is generated. RenderbufferStorageEXT deletes any existing data store for the renderbuffer and the contents of the data store after calling RenderbufferStorageEXT are undefined.

Sized Internal Format	Base Internal format	S Bits
STENCIL_INDEX1_EXT	STENCIL_INDEX	1
STENCIL_INDEX4_EXT	STENCIL_INDEX	4
STENCIL_INDEX8_EXT	STENCIL_INDEX	8
STENCIL_INDEX16_EXT	STENCIL_INDEX	16

Table 2.nnn Desired component resolution for each sized internal format that can be used only with renderbuffers.

A GL implementation may vary its allocation of internal component resolution based on any RenderbufferStorage parameter (except target), but the allocation and chosen internal format must not be a function of any other state and cannot be changed once they are established. The actual resolution in bits of each component of the allocated image can be queried with GetRenderbufferParameteriv as described in section 6.1.3.

4.4.2.2 Attaching Renderbuffer Images to a Framebuffer

A renderbuffer can be attached as one of the logical buffers of the currently bound framebuffer object by calling

```
    void FramebufferRenderbufferEXT(enum target,
                                    enum attachment,
                                    enum renderbuffertarget,
                                    uint renderbuffer);
```

<target> must be FRAMEBUFFER_EXT. INVALID_OPERATION is generated if the current value of FRAMEBUFFER_BINDING_EXT is zero when FramebufferRenderbufferEXT is called. <attachment> should be set to one of the attachment points of the framebuffer listed in table 1.nnn. <renderbuffertarget> must be RENDERBUFFER_EXT and <renderbuffer> should be set to the name of the renderbuffer object to be attached to the framebuffer. <renderbuffer> must be either zero or the name of an existing renderbuffer object of type <renderbuffertarget>, otherwise INVALID_OPERATION is generated. If <renderbuffer> is zero, then the value of <renderbuffertarget> is ignored.

If <renderbuffer> is not zero and if FramebufferRenderbufferEXT is successful, then the renderbuffer named <renderbuffer> will be used as the logical buffer identified by <attachment> of the framebuffer currently bound to <target>. The value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE_EXT for the specified attachment point is set to RENDERBUFFER_EXT and the value of FRAMEBUFFER_ATTACHMENT_OBJECT_NAME_EXT is set to <renderbuffer>. All other state values of the attachment point specified by <attachment> are set to their default values listed in table 5.nnn. No change is made to the state of the renderbuffer object and any previous attachment to the <attachment> logical buffer of the framebuffer object bound to framebuffer <target> is broken. If, on the other hand, the attachment is not successful, then no change is made to the state of either the renderbuffer object or the framebuffer object.

Calling FramebufferRenderbufferEXT with the <renderbuffer> name zero will detach the image, if any, identified by <attachment>, in the framebuffer currently bound to <target>. All state values of the attachment point specified by <attachment> in the object bound to <target> are set to their default values listed in table 5.nnn.

If a renderbuffer object is deleted while its image is attached to one or more attachment points in the currently bound framebuffer, then it is as if FramebufferRenderbufferEXT() had been called, with a <renderbuffer> of 0, for each attachment point to which this image was attached in the currently bound framebuffer. In other words, this renderbuffer image is first detached from all attachment points in the currently bound framebuffer. Note that the renderbuffer image is specifically *not* detached from any non-bound framebuffers. Detaching the image from any non-bound framebuffers is the responsibility of the application.

```
Name of attachment
---------------------------------------------------------------------------------
COLOR_ATTACHMENT0_EXT ... COLOR_ATTACHMENTn_EXT (where n is from 0 to MAX_COLOR_ATTACHMENTS_EXT-1)
DEPTH_ATTACHMENT_EXT
STENCIL_ATTACHMENT_EXT
---------------------------------------------------------------------------------
```

Table 1.nnn: "List of framebuffer attachment points"

4.4.2.3 Attaching Texture Images to a Framebuffer

GL supports copying the rendered contents of the framebuffer into the images of a texture object through the use of the routines CopyTexImage{1D|2D}, and CopyTexSubImage{1D|2D|3D}. Additionally, GL supports rendering directly into the images of a texture object.

To render directly into a texture image, a specified image from a texture object can be attached as one of the logical buffers of the currently bound framebuffer object by calling one of the following routines, depending on the type of the texture:

```
    void FramebufferTexture1DEXT(enum target, enum attachment,
                                 enum textarget, uint texture,
                                 int level);
    void FramebufferTexture2DEXT(enum target, enum attachment,
                                 enum textarget, uint texture,
                                 int level);
    void FramebufferTexture3DEXT(enum target, enum attachment,
                                 enum textarget, uint texture,
                                 int level, int zoffset);
```

In all three routines, <target> must be FRAMEBUFFER_EXT. INVALID_OPERATION is generated if the current value of FRAMEBUFFER_BINDING_EXT is zero when FramebufferTexture{1D|2D|3D}EXT is called. <attachment> must be one of the attachment points of the framebuffer listed in table 1.nnn.

In all three routines, if <texture> is zero, then <textarget>, <level>, and <zoffset> are ignored. If <texture> is not zero, then <texture> must either name an existing texture object with an target of <textarget>, or <texture> must name an existing cube map texture and <textarget> must be one of: TEXTURE_CUBE_MAP_POSITIVE_X, TEXTURE_CUBE_MAP_POSITIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z, TEXTURE_CUBE_MAP_NEGATIVE_X, TEXTURE_CUBE_MAP_NEGATIVE_Y, or TEXTURE_CUBE_MAP_NEGATIVE_Z. Otherwise, GL_INVALID_OPERATION is generated.

<level> specifies the mipmap level of the texture image to be attached to the framebuffer.

If <textarget> is TEXTURE_RECTANGLE_ARB, then <level> must be zero. If <textarget> is TEXTURE_3D, then <level> must be greater than or equal to zero and less than or equal to log base 2 of MAX_3D_TEXTURE_SIZE. If <textarget> is one of TEXTURE_CUBE_MAP_POSITIVE_X, TEXTURE_CUBE_MAP_POSITIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z, TEXTURE_CUBE_MAP_NEGATIVE_X, TEXTURE_CUBE_MAP_NEGATIVE_Y, or TEXTURE_CUBE_MAP_NEGATIVE_Z, then <level> must be greater than or equal to zero and less than or equal to log base 2 of MAX_CUBE_MAP_TEXTURE_SIZE. For all other values of <textarget>, <level> must be greater than or equal to zero and no larger than log base 2 of MAX_TEXTURE_SIZE. Otherwise, INVALID_VALUE is generated.

<zoffset> specifies the z-offset of a 2-dimensional image within a 3-dimensional texture. INVALID_VALUE is generated if <zoffset> is larger than MAX_3D_TEXTURE_SIZE-1.

For FramebufferTexture1DEXT, if <texture> is not zero, then <textarget> must be TEXTURE_1D.

For FramebufferTexture2DEXT, if <texture> is not zero, then <textarget> must be one of: TEXTURE_2D, TEXTURE_RECTANGLE_ARB, TEXTURE_CUBE_MAP_POSITIVE_X, TEXTURE_CUBE_MAP_POSITIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z, TEXTURE_CUBE_MAP_NEGATIVE_X, TEXTURE_CUBE_MAP_NEGATIVE_Y, or TEXTURE_CUBE_MAP_NEGATIVE_Z.

For FramebufferTexture3DEXT, if <texture> is not zero, then <textarget> must be TEXTURE_3D.

If <texture> is not zero, and if FramebufferTexture{1D|2D|3D}EXT is successful, then the specified texture image will be used as the logical buffer identified by <attachment> of the framebuffer currently bound to <target>. The value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE_EXT for the specified attachment point is set to TEXTURE and the value of FRAMEBUFFER_ATTACHMENT_OBJECT_NAME_EXT is set to <texture>. Additionally, the value of FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL for the named attachment point is set to <level>. If <texture> is a cubemap texture then, the value of FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE the named attachment point is set to <textarget>. If <texture> is a 3D texture, then the value of FRAMEBUFFER_ATTACHMENT_TEXTURE_3D_ZOFFSET for the named attachment point is set to <zoffset>. All other state values of the attachment point specified by <attachment> are set to their default values listed in table 5.nnn. No change is made to the state of the texture object, and any previous attachment to the <attachment> logical buffer of the framebuffer object bound to framebuffer <target> is broken. If, on the other hand, the attachment is not successful, then no change is made to the state of either the texture object or the framebuffer object.

Calling FramebufferTexture{1D|2D|3D}EXT with <texture> name zero will detach the image identified by <attachment>, if any, in the framebuffer currently bound to <target>. All state values of the attachment point specified by <attachment> are set to their default values listed in table 5.nnn.

If a texture object is deleted while its image is attached to one or more attachment points in the currently bound framebuffer, then it is as if FramebufferTexture{1D|2D|3D}EXT() had been called, with a <texture> of 0, for each attachment point to which this image was attached in the currently bound framebuffer. In other words, this texture image is first detached from all attachment points in the currently bound framebuffer. Note that the texture image is specifically *not* detached from any other framebuffer objects.

Detaching the texture image from any other framebuffer objects is the responsibility of the application.

4.4.3 Rendering When an Image of a Bound Texture Object is Also Attached to the Framebuffer

Special precautions need to be taken to avoid attaching a texture image to the currently bound framebuffer while the texture object is currently bound and enabled for texturing. Doing so could lead to the creation of a "feedback loop" between the writing of pixels by the GL's rendering operations and the simultaneous reading of those same pixels when used as texels in the currently bound texture. In this scenario, the framebuffer will be considered framebuffer complete (see section 4.4.4), but the values of fragments rendered while in this state will be undefined. The values of texture samples may be undefined as well, as described in section 3.8.8.

Specifically, the values of rendered fragments are undefined if all of the following conditions are true:

- an image from texture object <T> is attached to the currently bound framebuffer at attachment point <A>, and

- the texture object <T> is currently bound to a texture unit <U>, and

- the current fixed-function texture state or programmable vertex and/or fragment processing state makes it possible(*) to sample from the texture object <T> bound to texture unit <U>

while either of the following conditions are true:

- the value of TEXTURE MIN FILTER for texture object <T> is NEAREST or LINEAR, and the value of FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL_EXT for attachment point <A> is equal to the value of TEXTURE_BASE_LEVEL for the texture object <T>, or

- the value of TEXTURE_MIN_FILTER for texture object <T> is one of NEAREST_MIPMAP_NEAREST, NEAREST_MIPMAP LINEAR, LINEAR MIPMAP_NEAREST, or LINEAR_MIPMAP_LINEAR, and the value of FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL_EXT for attachment point <A> is within the the range specified by the current values of TEXTURE_BASE_LEVEL to q, inclusive, for the texture object <T>. (q is defined in the Mipmapping discussion of section 3.8.8),

(*) For the purpose of this discussion, we consider it "possible" to sample from the texture object <T> bound to texture unit <U>" if any of the following are true:

- programmable vertex and fragment processing is disabled and the target of texture object <T> is enabled according to the texture target precedence rules of section 3.8.15, or
- if FRAGMENT_PROGRAM_ARB is enabled and the currently bound fragment program contains any instructions that sample from the texture object <T> bound to <U>, or
- if the active fragment or vertex shader contains any instructions that might sample from the texture object <T> bound to <U> if even those instructions might only be executed conditionally.

Note that if TEXTURE_BASE_LEVEL and TEXTURE_MAX_LEVEL exclude any levels containing image(s) attached to the currently bound framebuffer, then the above conditions will not be met, (i.e., the above rule will not cause the values of rendered fragments to be undefined.)

4.4.4 Framebuffer Completeness

A framebuffer object is said to be "framebuffer complete" if all of its attached images, and all framebuffer parameters required to utilize the framebuffer for rendering and reading, are consistently defined and meet the requirements defined below. The rules of framebuffer completeness are dependent on the properties of the attached images, and on certain implementation dependent restrictions. A framebuffer must be complete to effectively be used as the destination for GL framebuffer rendering operations and the source for GL framebuffer read operations.

The internal formats of the attached images can affect the completeness of the framebuffer, so it is useful to first define the relationship between the internal format of an image and the attachment points to which it can be attached.

* The following base internal formats from table 3.15 are "color-renderable": RGB, RGBA, FLOAT_R_NV, FLOAT_RG_NV, FLOAT_RGB_NV, and FLOAT_RGBA_NV. The sized internal formats from table 3.16 that have a color-renderable base internal format are also color-renderable. No other formats, including compressed internal formats, are color-renderable.

* An internal format is "depth-renderable" if it is DEPTH_COMPONENT, or if it is one of the sized internal formats from table 3.16 that has a depth-renderable base internal format. No other formats are depth-renderable.

* An internal format is "stencil-renderable" if it is STENCIL_INDEX, or if it is one of the sized internal formats from table 2.nnn that has a stencil-renderable base internal format. No other formats are stencil-renderable.

4.4.4.1 Framebuffer Attachment Completeness

If the value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE_EXT for the framebuffer attachment point <attachment> is not NONE, then it is said that a framebuffer-attachable image, named <image>, is attached to the framebuffer at the attachment point. <image> is identified by the state in <attachment> as described in section 4.4.2.

The framebuffer attachment point <attachment> is said to be "framebuffer attachment complete" if the value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE_EXT for <attachment> is NONE (i.e., no image is attached), or if all of the following conditions are true:

* <image> is a component of an existing object with the name specified by FRAMEBUFFER_ATTACHMENT_OBJECT_NAME_EXT, and of the type specified by FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE_EXT.

* The width and height of <image> must be non-zero.

* If FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE_EXT is TEXTURE and FRAMEBUFFER_ATTACHMENT_OBJECT_NAME_EXT names a 3-dimensional texture, then FRAMEBUFFER_ATTACHMENT_TEXTURE_ZOFFSET_EXT must be smaller than the depth of the texture.

* If <attachment> is one of COLOR_ATTACHMENT0_EXT through COLOR_ATTACHMENTn_EXT, then <image> must have a color-renderable internal format.

* If <attachment> is DEPTH_ATTACHMENT_EXT, then <image> must have a depth-renderable internal format.

* If <attachment> is STENCIL_ATTACHMENT_EXT, then <image> must have a stencil-renderable internal format.

4.4.4.2 Framebuffer Completeness

In this subsection, each rule is followed by an error enum enclosed in { brackets }. Sections 4.4.4.2 and 4.4.4.3 explains the relevance of the error enums.

The framebuffer object <target> is said to be "framebuffer complete" if it is the window-system-provided framebuffer, or if all the following conditons are true:

* All framebuffer attachment points are "framebuffer attachment complete".
  { FRAMEBUFFER_INCOMPLETE_ATTACHMENT_EXT }

* There is at least one image attached to the framebuffer.
  { FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT_EXT }

* All attached images have the same width and height.
  { FRAMEBUFFER_INCOMPLETE_DIMENSIONS_EXT }

* All images attached to the attachment points COLOR_ATTACHMENT0_EXT through COLOR_ATTACHMENTn_EXT must have the same internal format.
  { FRAMEBUFFER_INCOMPLETE_FORMATS_EXT }

* The value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE_EXT must not be NONE for any color attachment point(s) named by DRAW_BUFFERi.
  { FRAMEBUFFER_INCOMPLETE_DRAW_BUFFER_EXT }

* If READ_BUFFER is not NONE, then the value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE_EXT must not be NONE for the color attachment point named by READ_BUFFER.
  { FRAMEBUFFER_INCOMPLETE_READ_BUFFER_EXT }

* The combination of internal formats of the attached images does not violate an implementation-dependent set of restrictions.
  { FRAMEBUFFER_UNSUPPORTED_EXT }

The enum in { brackets } after each clause of the framebuffer completeness rules specifies the return value of CheckFramebufferStatusEXT (see below) that is generated when that clause is violated. If more than one clause is violated, it is implementation-dependent exactly which enum will be returned by CheckFramebufferStatusEXT.

Performing any of the following actions may change whether the framebuffer is considered complete or incomplete.

- Binding to a different framebuffer with BindFramebufferEXT.

- Attaching an image to the framebuffer with FramebufferTexture{1D|2D|3D}EXT or FramebufferRenderbufferEXT.

- Detaching an image from the framebuffer with FramebufferTexture{1D|2D|3D}EXT or FramebufferRenderbufferEXT.

- Changing the width, height, or internal format of a texture image that is attached to the framebuffer by calling {Copy|Compressed}TexImage{1D|2D|3D}.

- Changing the width, height, or internal format of a renderbuffer that is attached to the framebuffer by calling RenderbufferStorageEXT.

- Deleting, with DeleteTextures or DeleteRenderbuffers, an object containing an image that is attached to a framebuffer object that is bound to the framebuffer.

- Changing READ_BUFFER or one of the DRAW_BUFFERS.

Although GL defines a wide variety of internal formats for framebuffer-attachable images, such as texture images and renderbuffer images, some implementations may not support rendering to particular combinations of internal formats. If the combination of formats of the images attached to a framebuffer object are not supported by the implementation, then the framebuffer is not complete under the clause labeled FRAMEBUFFER_UNSUPPORTED_EXT. There must exist, however, at least one combination of internal formats for which the framebuffer cannot be FRAMEBUFFER_UNSUPPORTED_EXT.

Because of the "implementation-dependent" clause of the framebuffer completeness test in particular, and because framebuffer completeness can change when the set of attached images is modified, it is strongly advised, though is not required, that an application check to see if the framebuffer is complete prior to rendering. The status of the framebuffer object currently bound to <target> can be queried by calling

```
    enum CheckFramebufferStatusEXT(enum target);
```

If <target> is not FRAMEBUFFER_EXT, INVALID_ENUM is generated. If CheckFramebufferStatusEXT is called within a Begin/End pair, INVALID_OPERATION is generated. If CheckFramebufferStatusEXT generates an error, 0 is returned.

Otherwise, an enum is returned that identifies whether or not the framebuffer bound to <target> is complete, and if not complete the enum identifies one of the rules of framebuffer completeness that is violated. If the framebuffer is complete, then FRAMEBUFFER_COMPLETE_EXT is returned.

4.4.4.3 Effects of Framebuffer Completeness on Framebuffer Operations

If the currently bound framebuffer is not framebuffer complete, then it is an error to attempt to use the framebuffer for writing or reading. This means that rendering commands such as Begin, RasterPos, any command that performs an implicit Begin, as well as commands that read the framebuffer such as ReadPixels and CopyTex{Sub}Image will generate the error INVALID_FRAMEBUFFER_OPERATION_EXT if called while the framebuffer is not framebuffer complete.

4.4.5 Effects of Framebuffer State on Framebuffer Dependent Values

The values of the state variables listed in table 9.nnn (Framebuffer Dependent Values) may change when a change is made to FRAMEBUFFER_BINDING_EXT, to the state of the currently bound framebuffer object, or to an image attached to the currently bound framebuffer object.

When FRAMEBUFFER_BINDING_EXT is zero, the values of the state variables listed in table 9.nnn are implementation defined.

When FRAMEBUFFER_BINDING_EXT is non-zero, if the currently bound framebuffer object is not framebuffer complete, then the values of the state variables listed in table 9.nnn are undefined.

When FRAMEBUFFER_BINDING_EXT is non-zero and the currently bound framebuffer object is framebuffer complete, then the values of the state variables listed in table 9.nnn are completely determined by FRAMEBUFFER_BINDING_EXT, the state of the currently bound framebuffer object, and the state of the images attached to the currently bound framebuffer object.

XXX [from jon leech] describe derivation of red green and blue size

4.4.6 Mapping between Pixel and Element in Attached Image

When FRAMEBUFFER_BINDING_EXT is non-zero, an operation that writes to the framebuffer modifies the image attached to the selected logical buffer, and an operation that reads from the framebuffer reads from the image attached to the selected logical buffer.

If the attached image is a renderbuffer image, then the window coordinates (x[w], y[w]) corresponds to the value in the renderbuffer image at the same coordinates.

If the attached image is a texture image, then the window coordinates (x[w], y[w]) correspond to the texel (i, j, k), from figure 3.10, as follows:

$$i = (x[w] - b)$$

$$j = (y[w] - b)$$

$$k = (zoffset - b)$$

where b is the texture image's border width, and zoffset is the value of FRAMEBUFFER_ATTACHMENT_TEXTURE_3D_ZOFFSET for the selected logical buffer. For a two-dimensional texture, k and zoffset are irrelevant; for a one-dimensional texture, j, k, and zoffset are both irrelevant.

(x[w], y[w]) corresponds to a border texel if x[w] or y[w] or zoffset is less than the border size, or if x[w] or y[w] or zoffset is greater than the border size plus the width or height or depth, resp., of the texture image.

Conversion to Framebuffer-Attachable Image Components

When an enabled color value is written to the framebuffer while FRAMEBUFFER_BINDING is non-zero, for each draw buffer the R, G, B, and A values are converted to internal components as described in table 3.15, according to the table row corresponding to the internal format of the framebuffer-attachable image attached to the selected logical buffer, and the resulting internal components are written to the image attached to logical buffer. The masking operations described in section 4.2.2 are also effective.

Conversion to RGBA Values

When a color value is read or is used as the source of a logical operation or blending, while FRAMEBUFFER_BINDING is non-zero, the components of the framebuffer-attachable image that is attached to the logical buffer selected by READ_BUFFER are first converted to R, G, B, and A values according to table 3.21 and the internal format of the attached image."

Additions to Chapter 5 of the OpenGL 1.5 Specification (Special Functions)

Added to section 5.4, as part of the discussion of which commands are not compiled into display lists:

"Certain commands, when called while compiling a display list, are not compiled into the display list but are executed immediately. These are: ..., GenFramebuffersEXT, BindFramebufferEXT, DeleteFramebuffersEXT, CheckFramebufferStatusEXT, GenRenderbuffersEXT, BindRenderbufferEXT, DeleteRenderbuffersEXT, RenderbufferStorageEXT, FramebufferTexture1DEXT, FramebufferTexture2DEXT, FramebufferTexture3DEXT, FramebufferRenderbufferEXT, GenerateMipmapEXT..."

Additions to Chapter 6 of the OpenGL 1.5 Specification (State and State Requests)

Add to section 6.1.3, Enumerated Queries:

In the list of state query functions, add:

```
"void GetFramebufferAttachmentParameterivEXT(enum target,
                                             enum attachment,
                                             enum pname,
                                             int *params);
```

<target> must be FRAMEBUFFER_EXT. <attachment> must be one of the attachment points of the framebuffer listed in table 1.nnn. <pname> must be one of the following: FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE_EXT, FRAMEBUFFER_ATTACHMENT_OBJECT_NAME_EXT, FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL_EXT, FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE_EXT, FRAMEBUFFER_ATTACHMENT_TEXTURE_3D_ZOFFSET_EXT.

If the framebuffer currently bound to <target> is zero, then INVALID_OPERATION is generated.

Upon successful return from GetFramebufferAttachmentParameterivEXT, if <pname> is FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE_EXT, then param will contain one of NONE, TEXTURE, or RENDERBUFFER_EXT, identifying the type of object which contains the attached image.

If the value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE_EXT is RENDERBUFFER_EXT, then

If <pname> is FRAMEBUFFER_ATTACHMENT_OBJECT_NAME_EXT, <params> will contain the name of the renderbuffer object which contains the attached image.

Otherwise, INVALID_ENUM is generated.

```
        If the value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE_EXT is
        TEXTURE, then

            If <pname> is FRAMEBUFFER_ATTACHMENT_OBJECT_NAME_EXT,
            then <params> will contain the name of the texture
            object which contains the attached image.

            If <pname> is FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL_EXT,
            then <params> will contain the mipmap level of the
            texture object which contains the attached image.

            If <pname> is
            FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE_EXT and the
            texture object named
            FRAMEBUFFER_ATTACHMENT_OBJECT_NAME_EXT is a cube map
            texture, then <params> will contain the cube map face of
            the cubemap texture object which contains the attached
            image.  Otherwise <params> will contain the value zero.

            If <pname> is
            FRAMEBUFFER_ATTACHMENT_TEXTURE_3D_ZOFFSET_EXT and the
            texture object named
            FRAMEBUFFER_ATTACHMENT_OBJECT_NAME_EXT is a
            3-dimensional texture, then <params> will contain the
            zoffset of the 2D image of the 3D texture object which
            contains the attached image.  Otherwise <params> will
            contain the value zero.

            Otherwise, INVALID_ENUM is generated.

    void GetRenderbufferParameterivEXT(enum target, enum pname,
                                       int* params);

        <target> must be RENDERBUFFER_EXT.  <pname> must be one of
        the symbolic values in table 8.nnn.

        If the renderbuffer currently bound to <target> is zero,
        then INVALID_OPERATION is generated.

        Upon successful return from GetRenderbufferParameterivEXT,
        if <pname> is RENDERBUFFER_WIDTH_EXT,
        RENDERBUFFER_HEIGHT_EXT, or
        RENDERBUFFER_INTERNAL_FORMAT_EXT, then <params> will contain
        the width in pixels, height in pixels, or internal format,
        respectively, of the image of the renderbuffer currently
        bound to <target>.

        Upon successful return from GetRenderbufferParameterivEXT,
        if <pname> is RENDERBUFFER_RED_SIZE_EXT,
        RENDERBUFFER_GREEN_SIZE_EXT, RENDERBUFFER_BLUE_SIZE_EXT,
        RENDERBUFFER_ALPHA_SIZE_EXT, RENDERBUFFER_DEPTH_SIZE_EXT, or
        RENDERBUFFER_STENCIL_SIZE_EXT, then <params> will contain
        the actual resolutions, (not the resolutions specified when
        the image array was defined), for the red, green, blue,
        alpha depth, or stencil components, respectively, of the
        image of the renderbuffer currently bound to <target>.
```

Otherwise, INVALID_ENUM is generated."

After section 6.1.13 and before section 6.1.14 (which should be renumbered 6.1.16), add two new sections:

6.1.14 Framebuffer Object Queries

The command

```
boolean IsFramebufferEXT( uint framebuffer );
```

returns TRUE if <framebuffer> is the name of an framebuffer object. If <framebuffer> is zero, or if <framebuffer> is a non-zero value that is not the name of an framebuffer object, IsFramebufferEXT return FALSE.

6.1.15 Renderbuffer Object Queries

The command

```
boolean IsRenderbufferEXT( uint renderbuffer );
```

returns TRUE if <renderbuffer> is the name of a renderbuffer object. If <renderbuffer> is zero, or if <renderbuffer> is a non-zero value that is not the name of a renderbuffer object, IsRenderbufferEXT return FALSE.

Errors

The error INVALID_OPERATION is generated if FRAMEBUFFER_BINDING_EXT is zero and DrawBuffer or DrawBuffers is called with a <buf> constant (other than NONE) that does not correspond to a buffer allocated to the GL by the window-system, including the constants COLOR_ATTACHMENT0_EXT through COLOR_ATTACHMENTn_EXT, where n is MAX_COLOR_ATTACHMENTS_EXT - 1.

The error INVALID_OPERATION is generated if FRAMEBUFFER_BINDING_EXT is non-zero and DrawBuffer, DrawBuffers, or ReadBuffer is called with a <buf> constant (other than NONE) that is not in the range COLOR_ATTACHMENT0_EXT through COLOR_ATTACHMENTn_EXT, where n is MAX_COLOR_ATTACHMENTS_EXT - 1.

The error INVALID_ENUM is generated if DrawBuffer or ReadBuffer is called with a <buf> constant that is not listed in table 4.4 or 10.nnn.

The error INVALID_ENUM is generated if DrawBuffers is called with a <buf> constant that is not listed in table 10.nnn or 11.nnn.

The error INVALID_FRAMEBUFFER_OPERATION_EXT is generated if the value of FRAMEBUFFER_STATUS_EXT is not FRAMEBUFFER_COMPLETE_EXT when any attempts to render to or read from the framebuffer are made.

The error INVALID_OPERATION is generated if GetFramebufferAttachmentParameterivEXT is called while the value of FRAMEBUFFER_BINDING_EXT is zero.

```
    The error INVALID_OPERATION is generated if
    FramebufferRenderbufferEXT or FramebufferTexture{1D|2D|3D}EXT is
    called  while the value of FRAMEBUFFER_BINDING_EXT is zero.

    The error INVALID_OPERATION is generated if RenderbufferStorageEXT
    or GetRenderbufferParameterivEXT is called while the value of
    RENDERBUFFER_BINDING_EXT is zero.

    The error INVALID_ENUM is generated if
    GetFramebufferAttachmentParameterivEXT is called with an
    <attachment> other than COLOR_ATTACHMENT0_EXT through
    COLOR_ATTACHMENTn_EXT, where n is MAX_COLOR_ATTACHMENTS_EXT - 1.

    The error INVALID_ENUM is generated if
    GetFramebufferAttachmentParameterivEXT is called with a <pname>
    other than FRAMEBUFFER_ATTACHMENT_OBJECT_NAME_EXT when the type of
    the attached object at the named attachment point is
    RENDERBUFFER_EXT.

    The error INVALID_ENUM is generated if
    GetFramebufferAttachmentParameterivEXT is called with a <pname>
    other than FRAMEBUFFER_ATTACHMENT_OBJECT_NAME_EXT,
    FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL_EXT,
    FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE_EXT, or
    FRAMEBUFFER_ATTACHMENT_TEXTURE_3D_ZOFFSET_EXT when the type of the
    attached object at the named attachment point is TEXTURE.

    The error INVALID_ENUM is generated if GetRenderbufferParameterivEXT
    is called with a <pname> other than RENDERBUFFER_WIDTH_EXT,
    RENDERBUFFER_HEIGHT_EXT, or RENDERBUFFER_INTERNAL_FORMAT_EXT,
    GL_RENDERBUFFER_RED_SIZE, GL_RENDERBUFFER_GREEN_SIZE,
    GL_RENDERBUFFER_BLUE_SIZE, GL_RENDERBUFFER_ALPHA_SIZE,
    GL_RENDERBUFFER_DEPTH_SIZE, or GL_RENDERBUFFER_STENCIL_SIZE.

    The error INVALID_VALUE is generated if RenderbufferStorageEXT is
    called with a <width> or <height> that is greater than
    MAX_RENDERBUFFER_SIZE_EXT.

    The error INVALID_ENUM is generated if RenderbufferStorageEXT is
    called with an <internalformat> that is not RGB, RGBA,
    DEPTH_COMPONENT, STENCIL_INDEX, or one of the internal formats from
    table 3.16 or table 2.nnn that has a base internal format of RGB,
    RGBA, DEPTH_COMPONENT, or STENCIL_INDEX.

    The error INVALID_OPERATION is generated if
    FramebufferRenderbufferEXT is called and <renderbuffer> is not the
    name of a renderbuffer object.

    The error INVALID_OPERATION is generated if
    FramebufferTexture{1D|2D|3D}EXT is called and <texture> is not the
    name of a texture object.

    The error INVALID_VALUE is generated if
    FramebufferTexture{1D|2D|3D}EXT is called with a <level> that is
    less than zero.
```

The error INVALID_VALUE is generated if FramebufferTexture2DEXT is called with a <level> that is not zero and <textarget> is TEXTURE_RECTANGLE_ARB.

The error INVALID_VALUE is generated if FramebufferTexture{1D|2D}EXT is called with a <level> that is greater than the log base 2 of MAX_TEXTURE_SIZE and <texture> is a 1D or 2D texture.

The error INVALID_VALUE is generated if FramebufferTexture2DEXT is called with a <level> that is greater than the log base 2 of MAX_CUBE_MAP_TEXTURE_SIZE and <texture> is a cubemap texture.

The error INVALID_VALUE is generated if FramebufferTexture3DEXT is called with a <level> greater than the log base 2 of the MAX_3D_TEXTURE_SIZE.

The error INVALID_VALUE is generated if FramebufferTexture3DEXT is called with a <zoffset> that is larger than MAX_3D_TEXTURE_SIZE-1.

The error INVALID_ENUM is generated if CheckFramebufferStatusEXT is called and <target> is not FRAMEBUFFER_EXT.

The error INVALID_OPERATION is generated if CheckFramebufferStatusEXT is called within a Begin/End pair.

The error OUT_OF_MEMORY is generated if the GL is unable to create a data store of the required size when calling RenderbufferStorageEXT.

The error INVALID_OPERATION is generated if GenerateMipmapEXT is called with a <target> of TEXTURE_CUBE_MAP and the texture object currently bound to TEXTURE_CUBE_MAP is not "cube complete" as defined in section 3.8.10

New State

(add new table 3.nnn, "Framebuffer (state per framebuffer target binding point)")

Get Value	Type	Get Command	Initial Value	Description	Section	Attribute
FRAMEBUFFER_BINDING_EXT	Z	GetIntegerv	0	name of framebuffer object bound to FRAMEBUFFER_EXT target	4.4.1	-

(insert new table 4.nnn, "Framebuffer (state per framebuffer object)")

Get Value	Type	Get Command	Initial Value	Description	Section	Attribute
DRAW_BUFFERi [1]	1 + xZ(10*)	GetIntegerv	see 4.2.1	draw buffer selected for color output i	4.2.1	color-buffer
READ_BUFFER [2]	Z(3)	GetIntegerv	see 4.3.2	read source	4.3.2	pixel

[1] prior to this extension, the DRAW_BUFFERi state was described in table 6.21 "Framebuffer Control" (of OpenGL 2.0 spec)
[2] prior to this extension, the READ_BUFFER state was described in table 6.26 "Pixel" (of OpenGL 2.0 spec)

(insert new table 5.nnn, "Framebuffer (state per framebuffer object attachment point)")

Get Value	Type	Get Command	Initial Value	Description	Section	Attribute
FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE_EXT	Z	GetFramebufferAttachmentParameterivEXT	NONE	type of image attached to framebuffer attachment point	4.4.2.2 and 4.4.2.3	-
FRAMEBUFFER_ATTACHMENT_OBJECT_NAME_EXT	Z	GetFramebufferAttachmentParameterivEXT	0	name of object attached to framebuffer attachment point	4.4.2.2 and 4.4.2.3	-
FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL_EXT	Z	GetFramebufferAttachmentParameterivEXT	0	mipmap level of texture image attached, if object attached is texture.	4.4.2.2 and 4.4.2.3	-
FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE_EXT	Z+	GetFramebufferAttachmentParameterivEXT	TEXTURE_ CUBE_MAP_ POSITIVE_X	cubemap face of texture image attached, if object attached is cubemap texture.	4.4.2.2 and 4.4.2.3	-
FRAMEBUFFER_ATTACHMENT_TEXTURE_3D_ZOFFSET_EXT	Z	GetFramebufferAttachmentParameterivEXT	0	zoffset of texture image attached, if object attached is 3D texture.	4.4.2.2 and 4.4.2.3	-

(insert new table 7.nnn, "Renderbuffers (state per renderbuffer target and binding point)")

Get Value	Type	Get Command	Initial Value	Description	Section	Attribute
RENDERBUFFER_BINDING_EXT	Z	GetIntegerv	0	renderbuffer object bound to RENDERBUFFER_EXT	4.4.2.1	-

(insert new table 8.nnn, "Renderbuffers (state per renderbuffer object)")

Get Value	Type	Get Command	Initial Value	Description	Section	Attribute
RENDERBUFFER_WIDTH_EXT	Z	GetRenderbufferParameterivEXT	0	width of renderbuffer	4.4.2.1	-
RENDERBUFFER_HEIGHT_EXT	Z	GetRenderbufferParameterivEXT	0	height of renderbuffer	4.4.2.1	-
RENDERBUFFER_INTERNAL_FORMAT_EXT	Z+	GetRenderbufferParameterivEXT	RGBA	internal format of renderbuffer	4.4.2.1	-
RENDERBUFFER_RED_SIZE_EXT	Z	GetRenderbufferParameterivEXT	0	size in bits of renderbuffer image's red component	4.4.2.1	-
RENDERBUFFER_GREEN_SIZE_EXT	Z	GetRenderbufferParameterivEXT	0	size in bits of renderbuffer image's green component	4.4.2.1	-
RENDERBUFFER_BLUE_SIZE_EXT	Z	GetRenderbufferParameterivEXT	0	size in bits of renderbuffer image's blue component	4.4.2.1	-
RENDERBUFFER_ALPHA_SIZE_EXT	Z	GetRenderbufferParameterivEXT	0	size in bits of renderbuffer image's alpha component	4.4.2.1	-
RENDERBUFFER_DEPTH_SIZE_EXT	Z	GetRenderbufferParameterivEXT	0	size in bits of renderbuffer image's depth component	4.4.2.1	-
RENDERBUFFER_STENCIL_SIZE_EXT	Z	GetRenderbufferParameterivEXT	0	size in bits of renderbuffer image's stencil component	4.4.2.1	-

Move the following existing state from "Implementation Dependent Values", tables 6.31-6.36 to into a new table called "Framebuffer Dependent Values", table 9.nnn.

```
    Get Value
    ---------
    AUX_BUFFERS
    MAX_DRAW_BUFFERS
    RGBA_MODE
    INDEX_MODE
    DOUBLEBUFFER
    STEREO
    SAMPLE_BUFFERS
    SAMPLES
    RED_BITS
    GREEN_BITS
    BLUE_BITS
    ALPHA_BITS
    DEPTH_BITS
    STENCIL_BITS
    ACCUM_RED_BITS
    ACCUM_GREEN_BITS
    ACCUM_BLUE_BITS
    ACCUM_ALPHA_BITS
    STENCIL_REF
```

To the same table called "Framebuffer Dependent Values", table 9.nnn add the following new framebuffer dependent state.

Get Value	Type	Get Command	Minimum Value	Description	Section	Attribute
MAX_COLOR_ATTACHMENTS_EXT	Z+	GetIntegerv	1	Maximum number of attachment points for color buffers when using framebuffer objects	4.4.2.2	-

New Implementation Dependent State

Get Value	Type	Get Command	Minimum Value	Description	Section	Attribute
MAX_RENDERBUFFER_SIZE_EXT	Z+	GetIntegerv	1	Maximum width and height of renderbuffers supported by the implementation	4.4.2.1	-

Additions to the AGL/GLX/WGL Specifications and dependencies on WGL_ARB_make_current_read, GLX_SGI_make_current_read, and GLX 1.3

The color, depth, stencil, aux, and accum logical buffers defined by the <draw> and <read> drawables passed to glXMakeContextCurrent, glXMakeCurrent, and glXMakeCurrentRead are ignored while the value of FRAMEBUFFER_BINDING_EXT is non-zero.

Dependencies on ATI_draw_buffers and ARB_draw_buffers

If neither ATI_draw_buffers nor ARB_draw_buffers are supported, then all discussions of DrawBuffers should be ignored.

```
    In addition, the language describing DrawBuffers are derived from a
    combination of the ARB_draw_buffers specification and section 4.2.1
    of the OpenGL 2.0 specification.
```

Dependencies on ARB_fragment_program, ARB_fragment_shader, and ARB_vertex_shader

```
    If ARB_fragment_program, ARB_fragment_shader, and ARB_vertex_shader
    are all not supported, then all references to the currently bound
    program or shader should be ignored.
```

Dependencies on ARB_texture_rectangle

```
    If ARB_texture_rectangle is not supported, then all references to
    TEXTURE_RECTANGLE_ARB should be ignored.
```

Dependencies on EXT_packed_depth_stencil

```
    If EXT_packed_depth_stencil is not supported, then all references to
    DEPTH_STENCIL internal formats should be ignored.
```

Dependencies on NV_float_buffer

```
    If NV_float_buffer is not supported, then all references to the
    following internal formats should be ignored: FLOAT_R_NV,
    FLOAT_RG_NV, FLOAT_RGB_NV, and FLOAT_RGBA_NV.
```

Dependencies on NV_texture_shader

```
    The following base internal formats are not color-renderable,
    depth-renderable, or stencil-renderable: HILO_NV, DSDT_NV,
    DSDT_MAG_NV, and DSDT_MAG_INTENSITY_NV.
```

GLX Protocol

```
        Seventeen new GL commands are added.

        The following ten rendering commands are sent to the sever as part
        of a glXRender request:
```

BindRenderbufferEXT

```
            2        12               rendering command length
            2        4316             rendering command opcode
            4        ENUM             target
            4        CARD32           renderbuffer
```

DeleteRenderbufferEXT

```
            2        8+n*4            rendering command length
            2        4317             rendering command opcode
            4        CARD32           n
            n*4      LISTofCARD32     renderbuffers
```

RenderbufferStorageEXT

2	20	rendering command length
2	4318	rendering command opcode
4	ENUM	target
4	ENUM	internalFormat
4	CARD32	width
4	CARD32	height

BindFramebufferEXT

2	12	rendering command length
2	4319	rendering command opcode
4	ENUM	target
4	CARD32	framebuffer

DeleteFramebufferEXT

2	8+n*4	rendering command length
2	4320	rendering command opcode
4	CARD32	n
n*4	LISTofCARD32	framebuffers

FramebufferTexture1DEXT

2	24	rendering command length
2	4321	rendering command opcode
4	ENUM	target
4	ENUM	attachement
4	ENUM	textarget
4	CARD32	texture
4	CARD32	level

FramebufferTexture2DEXT

2	24	rendering command length
2	4322	rendering command opcode
4	ENUM	target
4	ENUM	attachement
4	ENUM	textarget
4	CARD32	texture
4	CARD32	level

FramebufferTexture3DEXT

2	28	rendering command length
2	4323	rendering command opcode
4	ENUM	target
4	ENUM	attachement
4	ENUM	textarget
4	CARD32	texture
4	CARD32	level
4	CARD32	zoffset

FramebufferRenderbufferEXT

2	20	rendering command length
2	4324	rendering command opcode
4	ENUM	target
4	ENUM	attachment
4	ENUM	renderbuffertarget
4	CARD32	renderbuffer

```
GenerateMipmapEXT
    2        8                rendering command length
    2        4325             rendering command opcode
    4        ENUM             target

The remaining seven commands are non-rendering commands.  These
commands are sent separately (i.e., not as part of a glXRender or
glXRenderLarge request), using the glXVendorPrivateWithReply
request:

IsRenderbufferEXT
    1        CARD8            opcode (X assigned)
    1        17               GLX opcode (X_GLXVendorPrivateWithReply)
    2        4                request length
    4        1422             vendor specific opcode
    4        GLX_CONTEXT_TAG  context tag
    4        CARD32           renderbuffer
  =>
    1        1                reply
    1                         unused
    2        CARD16           sequence number
    4        0                reply length
    4        BOOL32           return value
    20                        unused

GenRenderbuffersEXT
    1        CARD8            opcode (X assigned)
    1        17               GLX opcode (X_GLXVendorPrivateWithReply)
    2        4                request length
    4        1423             vendor specific opcode
    4        GLX_CONTEXT_TAG  context tag
    4        CARD32           n
  =>
    1        1                reply
    1                         unused
    2        CARD16           sequence number
    4        m                reply length
    4                         unused
    4        CARD32           n
    16                        unused
    n*4      LISTofCARD32     renderbuffers
```

```
        GetRenderbufferParameterivEXT
            1        CARD8            opcode (X assigned)
            1        17               GLX opcode (X_GLXVendorPrivateWithReply)
            2        5                request length
            4        1424             vendor specific opcode
            4        GLX_CONTEXT_TAG  context tag
            4        ENUM             target
            4        ENUM             pname
          =>
            1        1                reply
            1                         unused
            2        CARD16           sequence number
            4        m                reply length, m = (n == 1 ? 0 : n)
            4                         unused
            4        CARD32           n

            if (n = 1) this follows:

            4        CARD32           params
            12                        unused

            otherwise this follows:

            16                        unused
            n*4      LISTofCARD32     params

        IsFramebufferEXT
            1        CARD8            opcode (X assigned)
            1        17               GLX opcode (X_GLXVendorPrivateWithReply)
            2        4                request length
            4        1425             vendor specific opcode
            4        GLX_CONTEXT_TAG  context tag
            4        CARD32           framebuffer
          =>
            1        1                reply
            1                         unused
            2        CARD16           sequence number
            4        0                reply length
            4        BOOL32           return value
            20                        unused

        GenFramebuffersEXT
            1        CARD8            opcode (X assigned)
            1        17               GLX opcode (X_GLXVendorPrivateWithReply)
            2        4                request length
            4        1426             vendor specific opcode
            4        GLX_CONTEXT_TAG  context tag
            4        CARD32           n
          =>
            1        1                reply
            1                         unused
            2        CARD16           sequence number
            4        n                reply length
            4                         unused
            4        CARD32           n
            16                        unused
            n*4      LISTofCARD32     framebuffers
        CheckFramebufferStatusEXT
            1        CARD8           opcode (X assigned)
            1        17              GLX opcode (X_GLXVendorPrivateWithReply)
            2        4               request length
            4        1427            vendor specific opcode
            4        GLX_CONTEXT_TAG context tag
            4        ENUM            target
          =>
            1        1               reply
            1                        unused
            2        CARD16          sequence number
            4        0               reply length
            4        ENUM            return value
            20                       unused

        GetFramebufferAttachementParameterivEXT
            1        CARD8           opcode (X assigned)
            1        17              GLX opcode (X_GLXVendorPrivateWithReply)
            2        6               request length
            4        1428            vendor specific opcode
            4        GLX_CONTEXT_TAG context tag
            4        ENUM            target
            4        ENUM            attachment
            4        ENUM            pname
          =>
            1        1               reply
            1                        unused
            2        CARD16          sequence number
            4        m               reply length, m = (n == 1 ? 0 : n)
            4                        unused
            4        CARD32          n

            if (n = 1) this follows:

            4        CARD32          params
            12                       unused

            otherwise this follows:

            16                       unused
            n*4      LISTofCARD32    params
```

Usage Examples

The following examples use a helper macro for CHECK_FRAMEBUFFER_STATUS, defined below.

Example (6) gives a (very slightly) more robust example of handling the possible return values for glCheckFramebufferStatusEXT.

```
#define CHECK_FRAMEBUFFER_STATUS()                            \
  {                                                           \
    GLenum status;                                            \
    status = glCheckFramebufferStatusEXT(GL_FRAMEBUFFER_EXT); \
    switch(status) {                                          \
      case GL_FRAMEBUFFER_COMPLETE_EXT:                       \
        break;                                                \
      case GL_FRAMEBUFFER_UNSUPPORTED_EXT:                    \
        /* choose different formats */                        \
        break;                                                \
      default:                                                \
        /* programming error; will fail on all hardware */    \
        assert(0);                                            \
    }
  }
```

(1) Render to 2D texture with a depth buffer

```
    // Given:  color_tex - TEXTURE_2D color texture object
    //         depth_rb  - GL_DEPTH renderbuffer object
    //         fb        - framebuffer object

    // Enable render-to-texture
    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);

    // Set up color_tex and depth_rb for render-to-texture
    glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                              GL_COLOR_ATTACHMENT0_EXT,
                              GL_TEXTURE_2D, color_tex, 0);
    glFramebufferRenderbufferEXT(GL_FRAMEBUFFER_EXT,
                                 GL_DEPTH_ATTACHMENT_EXT,
                                 GL_RENDERBUFFER_EXT, depth_rb);

    // Check framebuffer completeness at the end of initialization.
    CHECK_FRAMEBUFFER_STATUS();

    <draw to the texture and renderbuffer>

    // Re-enable rendering to the window
    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);

    glBindTexture(GL_TEXTURE_2D, color_tex);
    <draw to the window, reading from the color_tex>
```

(2) Application that supports both RBBCTT (render back buffer, copy to texture) and RTT (render to texture). The migration path from RBBCTT to RTT is easy.

```
    if (useFramebuffer) {
        glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);
        glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                                  GL_COLOR_ATTACHMENT0_EXT,
                                  GL_TEXTURE_2D, color_tex, 0);
        CHECK_FRAMEBUFFER_STATUS();
    }

    draw_to_texture();

    glBindTexture (GL_TEXTURE_2D, color_tex);
    if (useFramebuffer) {
        glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
    } else { // copy tex path
        glCopyTexSubImage(...);
    }
```

(3) Simple render-to-texture loop with initialization. Create an RGB8 texture, a 24-bit depth renderbuffer, and a stencil renderbuffer. In a loop, alternate between rendering to, and texturing out of, the color texture.

```
    glGenFramebuffersEXT(1, &fb);
    glGenTextures(1, &color_tex);
    glGenRenderbuffersEXT(1, &depth_rb);
    glGenRenderbuffersEXT(1, &stencil_rb);

    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);

    // initialize color texture
    glBindTexture(GL_TEXTURE_2D, color_tex);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB8, 512, 512, 0,
                 GL_RGB, GL_INT, NULL);
    glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                              GL_COLOR_ATTACHMENT0_EXT,
                              GL_TEXTURE_2D, color_tex, 0);

    // initialize depth renderbuffer
    glBindRenderbufferEXT(GL_RENDERBUFFER_EXT, depth_rb);
    glRenderbufferStorageEXT(GL_RENDERBUFFER_EXT,
                             GL_DEPTH_COMPONENT24, 512, 512);
    glFramebufferRenderbufferEXT(GL_FRAMEBUFFER_EXT,
                                 GL_DEPTH_ATTACHMENT_EXT,
                                 GL_RENDERBUFFER_EXT, depth_rb);

    // initialize stencil renderbuffer
    glBindRenderbufferEXT(GL_RENDERBUFFER_EXT, stencil_rb);
    glRenderbufferStorageEXT(GL_RENDERBUFFER_EXT,
                             GL_STENCIL_INDEX, 512, 512);
    glFramebufferRenderbufferEXT(GL_FRAMEBUFFER_EXT,
                                 GL_STENCIL_ATTACHMENT_EXT,
                                 GL_RENDERBUFFER_EXT, stencil_rb);

    // Check framebuffer completeness at the end of initialization.
    CHECK_FRAMEBUFFER_STATUS();

    loop {
        glBindTexture(GL_TEXTURE_2D, 0);
        glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);

        <draw to the texture>

        glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
        glBindTexture(GL_TEXTURE_2D, color_tex);

        <draw to the window, reading from the color texture>
    }
```

(4) Render-to-texture loop with automatic mipmap generation. There are N framebuffers, N mipmap color textures, and a single shared depth renderbuffer. The depth renderbuffer is not a mipmap.

```
GLuint fb_array[N];
GLuint color_tex_array[N];
GLuint depth_rb;

glGenFramebuffersEXT(N, fb_array);
glGenTextures(N, color_tex_array);
glGenRenderbuffersEXT(1, &depth_rb);

// initialize color textures
for (int i=0; i<N; i++) {
  glBindTexture(GL_TEXTURE_2D, color_tex_array[N]);
  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB8, 512, 512, 0,
               GL_RGB, GL_INT, NULL);

  // establish a mipmap chain for the texture
  glGenerateMipmapEXT(GL_TEXTURE_2D);
}

// initialize depth renderbuffer
glBindRenderbufferEXT(GL_RENDERBUFFER_EXT, depth_rb);
glRenderbufferStorageEXT(GL_RENDERBUFFER_EXT,
                         GL_DEPTH_COMPONENT24, 512, 512);

// setup framebuffers, sharing depth
for (int i=0; i<N; i++) {
  glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb_array[i]);
  glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                            GL_COLOR_ATTACHMENT0_EXT,
                            GL_TEXTURE_2D, color_tex_array[i], 0);
  glFramebufferRenderbufferEXT(GL_FRAMEBUFFER_EXT,
                               GL_DEPTH_ATTACHMENT_EXT,
                               GL_RENDERBUFFER_EXT, depth_rb);
}

// Check framebuffer completeness at the end of initialization.
CHECK_FRAMEBUFFER_STATUS();
```

```
        loop {
            glBindTexture(GL_TEXTURE_2D, 0);

            for (int i=0; i<N; i++) {
              glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb_array[i]);
              <draw to texture i>
            }

            glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);

            // automatically generate mipmaps
            for (int i=0; i<N; i++) {
              glBindTexture(GL_TEXTURE_2D, color_tex_array[i]);
              glGenerateMipmapEXT(GL_TEXTURE_2D);
            }

            <draw to the window, reading from the color textures>
        }
```

**(5) Render-to-texture loop with custom mipmap generation.
The depth renderbuffer is not a mipmap.**

```
        glGenFramebuffersEXT(1, &fb);
        glGenTextures(1, &color_tex);
        glGenRenderbuffersEXT(1, &depth_rb);

        glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);

        // initialize color texture and establish mipmap chain
        glBindTexture(GL_TEXTURE_2D, color_tex);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB8, 512, 512, 0,
                     GL_RGB, GL_INT, NULL);
        glGenerateMipmapEXT(GL_TEXTURE_2D);
        glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                                  GL_COLOR_ATTACHMENT0_EXT,
                                  GL_TEXTURE_2D, color_tex, 0);

        // initialize depth renderbuffer
        glBindRenderbufferEXT(GL_RENDERBUFFER_EXT, depth_rb);
        glRenderbufferStorageEXT(GL_RENDERBUFFER_EXT,
                                 GL_DEPTH_COMPONENT24, 512, 512);
        glFramebufferRenderbufferEXT(GL_FRAMEBUFFER_EXT,
                                     GL_DEPTH_ATTACHMENT_EXT,
                                     GL_RENDERBUFFER_EXT, depth_rb);

        // Check framebuffer completeness at the end of initialization.
        CHECK_FRAMEBUFFER_STATUS();
```

```
loop {
    glBindTexture(GL_TEXTURE_2D, 0);

    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);
    glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                              GL_COLOR_ATTACHMENT0_EXT,
                              GL_TEXTURE_2D, color_tex, 0);
    glFramebufferRenderbufferEXT(GL_FRAMEBUFFER_EXT,
                                 GL_DEPTH_ATTACHMENT_EXT,
                                 GL_RENDERBUFFER_EXT, depth_rb);

    <draw to the base level of the color texture>

    // custom-generate successive mipmap levels
    glFramebufferRenderbufferEXT(GL_FRAMEBUFFER_EXT,
                                 GL_DEPTH_ATTACHMENT_EXT,
                                 GL_RENDERBUFFER_EXT, 0);
    glBindTexture(GL_TEXTURE_2D, color_tex);
    foreach (level > 0, in order of increasing values of level) {
        glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                                  GL_COLOR_ATTACHMENT0_EXT,
                                  GL_TEXTURE_2D, color_tex, level);
        glTexParameteri(TEXTURE_2D, TEXTURE_BASE_LEVEL, level-1);
        glTexParameteri(TEXTURE_2D, TEXTURE_MAX_LEVEL, level-1);

        <draw to level>
    }
    glTexParameteri(TEXTURE_2D, TEXTURE_BASE_LEVEL, 0);
    glTexParameteri(TEXTURE_2D, TEXTURE_MAX_LEVEL, max);

    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
    <draw to the window, reading from the color texture>
}
```

(6) Pseudo-code example of one method of responding to FRAMEBUFFER_UNSUPPORTED_EXT

```
        bool done = false;
        bool success = false;
        int  configurationNumber = 0;
        GLenum status;

        while (!done)
        {
            for (each framebuffer-attachable image)
            {

ChooseInternalFormatForFramebufferAttachableImage(configurationNumber);

                CreateFramebufferAttachableImage();

                AttachFramebufferAttachableImageToFramebuffer();
            }

            status = glCheckFramebufferStatusEXT(GL_FRAMEBUFFER_EXT);
            switch(status)
            {
                case GL_FRAMEBUFFER_COMPLETE_EXT:
                    success = true;
                    done = true;
                    break;

                case GL_FRAMEBUFFER_UNSUPPORTED_EXT:
                    if (configCount < MAX_NUM_CONFIGS_I_WANT_TO_TRY)
                    {
                        printf("current config not supported, trying again);
                        configurationNumber++;
                    }
                    else
                    {
                        printf("couldn't find a supported config\n");
                        success = false;
                        done = true;
                    }
                    break;

                default:
                    // programming error; will fail on all hardware
                    FatalError();
                    exit(1);
            }
        }
        if (!success)
        {
            printf("couldn't find a supported config\n");
            FatalError();
            exit(1);
        }

        // Current framebuffer is supported and complete!!
        Draw();
```

(7) Render to depth texture with no color attachments

```
        // Given:  depth_tex - TEXTURE_2D depth texture object
        //         fb        - framebuffer object

        // Enable render-to-texture
        glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);

        // Set up depth_tex for render-to-texture
        glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                                  GL_DEPTH_ATTACHMENT_EXT,
                                  GL_TEXTURE_2D, depth_tex, 0);

        // No color buffer to draw to or read from
        glDrawBuffer(GL_NONE);
        glReadBuffer(GL_NONE);

        // Check framebuffer completeness at the end of initialization.
        CHECK_FRAMEBUFFER_STATUS();

        <draw something>

        // Re-enable rendering to the window
        glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);

        glBindTexture(GL_TEXTURE_2D, depth_tex);
        <draw to the window, reading from the depth_tex>
```

(8) FBO and ARB_draw_buffers

```
    // Given: color_texA - TEXTURE_2D color texture object
    // Given: color_texB - TEXTURE_2D color texture object
    //        depth_rb   - GL_DEPTH renderbuffer object
    //        fb         - framebuffer object

    // Set up the framebuffer object
    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);
    glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                              GL_COLOR_ATTACHMENT0_EXT,
                              GL_TEXTURE_2D, color_texA, 0);
    glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                              GL_COLOR_ATTACHMENT1_EXT,
                              GL_TEXTURE_2D, color_texB, 0);
    glFramebufferRenderbufferEXT(GL_FRAMEBUFFER_EXT,
                                 GL_DEPTH_ATTACHMENT_EXT,
                                 GL_RENDERBUFFER_EXT, depth_rb);

    // Enable both attachments as draw buffers
    GLenum drawbuffers = {GL_COLOR_ATTACHMENT0_EXT,
                          GL_COLOR_ATTACHMENT1_EXT};
    glDrawBuffers(2, drawbuffers);

    // Check framebuffer completeness at the end of initialization.
    CHECK_FRAMEBUFFER_STATUS();

    // Enable fragment program that writes to both gl_FragData[0]
    // and gl_FragData[1]

    <draw something>

    // Disable fragment program

    // Re-enable rendering to the window
    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);

    // Bind both textures, each to a different texture unit
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, color_texA);
    glActiveTexture(GL_TEXTURE1);
    glBindTexture(GL_TEXTURE_2D, color_texB);

    <draw to the window>
```

Issues

```
    (1)  We obviously won't call this "ARB_compromise_buffers", so
         what name should we use?

            RESOLUTION: resolved, EXT_framebuffer_object

            Possibilities considered include:
                EXT_framebuffer
                EXT_framebuffer_object
                EXT_renderable_buffers
                EXT_renderbuffer
                EXT_superbuffers (hah!)
                EXT_renderable_image
                EXT_render_image

            The lead candidates were EXT_renderable_image and
            EXT_framebuffer_object Since this extension introduced both
            new concepts into OpenGL, this was a bit of a toss up.
            EXT_framebuffer_object was chosen based on a weak precedent
            given by EXT_texture_object and ARB_vertex_buffer_object

    (2)  Many developers complain about the OpenGL/glX/WGL/agl pbuffer
         API, which they use both to do "render to texture" and to do
         general offscreen (non-windowed) accelerated rendering.  This
         extension is intended to subsume, some and perhaps all of, the
         functionality currently handled by pbuffers.  Should this
         extension (initially?) support only render-to-texture or should
         it try to provide an OpenGL API to fully replace the pbuffer
         API?

            RESOLUTION:  This extension should fully replace the pbuffer API.

            The implication of this decision is that this API should provide
            a way to support rendering to offscreen buffers that are not
            textures.

    (3)  As a consequence of issue (2), this extension adds the concept of
         share-able, non-texturable renderable entitites that can be
         used as color buffers, depth buffers, stencil buffers, etc.
         The OpenGL spec refers to these entities as "logical buffers".
         What should this spec call them?

            RESOLUTION: "renderbuffer", (one word)

            We could just call them "logical buffers", but is there a
            better name?
```

```
        The group considered:
        logical buffer  - possible, kind of general
        render buffer   - clear, (one word or two?)
        renderable      - clear, but may conflict with glx "drawable"
        drawable        - confusing: glx "drawable" == gl "framebuffer"
        render surface  - possible
        render target   - possible
        image buffer    - may get confused with Tex"Image"
        image           - may get confused with Tex"Image"
        surface buffer  - too verbose?
        surface         - too general
        others???

        The group felt "render buffer " (or possibly "renderbuffer")
        provides for the clearest expression of the purpose for
        these buffers.

        We finally decided on "renderbuffer" because we didn't want
        to use "render" as an adjective to describe a generic
        buffer, but rather decided to coin a new compound word to
        describe this concept.
```

(4) How should the specification refer to the group of various types of objects that can be attached to the framebuffer attachment points?

```
        RESOLUTION:  The specification will use the phrase
        "framebuffer-attachable images" to mean the 2D array of
        pixels (image) of a "renderbuffer", a "texture", or any
        other items that could be attached to a framebuffer.

            Options considered include:
              "render target"
              "renderable image"
              "framebuffer-attachable

        Initially, we chose the phrase "render target" for this but
        felt it didn't accurately capture the concept of a 2D array
        of pixels that was simultaneously useable as the storage of
        a texture object and the destination of rendering.

        We then tried to borrow the "image" language of OpenGL which
        describes texture's pixel arrays as "images" and we chose
        the term "renderable image".

        However, in the end, we felt that the salient characteristic
        of these images was that we could attach them to a
        framebuffer and settled on the term "framebuffer-attachable
        image".
```

(5) *How should the specification refer to the places in a framebuffer that can hold a framebuffer-attachable image?*

RESOLUTION: This state is called an "attachment point" of the framebuffer.

"attachment points" will be be used to describe the framebuffer state that holds a connection to a given framebuffer-attachable image (a renderbuffer image or a texture image). The framebuffer attachment points include the framebuffer's color buffers, stencil buffer, depth buffer, and aux buffer.

The word "attach" is being used to refer to connecting one object to another. "bind" refers to connecting an object to the context state. A texture image can be attached to a framebuffer object, but a framebuffer object is bound into the context state vector.

(6) *This extension adds the concept of collections of "logical buffers", to replace the window-system provided collection (drawable, or window) of logical buffers. What should we call these?*

RESOLUTION: "framebuffer"

For the "collection of logical buffers" object, the group considered the names: "framebuffer", "renderTarget", "drawable". We chose "framebuffer" since this is consistent with how the OpenGL specification already uses the word framebuffer.

(7) *This extension introduces two new object types into the OpenGL: renderbuffer objects and framebuffer objects. For handling these objects, there are two main object manipulation methodology precedents to choose from:*

```
1) "texture/program/vbo" object model:
        app-supplied int handles,
        Gen/is/Bind/Delete functions

2) "GLSL" object model:
        driver-supplied GLhandle handles,
        Create/Delete/Attach, etc
```

Which methodology should this extension use for each new object?

RESOLUTION: Use Option (1), "texture" object methodology, for both "renderbuffer" objects and framebuffer objects.

This is consistent with the June, 2004 ARB meeting vote to use the "texture" object methodlogy as the default object methodology.

(8) Do we need separate framebuffer objects?

RESOLUTION: yes.

The framebuffer object is an object to encapsulate the state of the framebuffer and the collection of framebuffer-attachable images attached to the logical buffer attachment points. A question was raised early on about whether we should have separate, shareable framebuffer objects or we should fold a single framebuffer "object" state vector into the context.

We decided to leave framebuffer objects in the API, with the understanding that we could easily remove them from the API and the spec later if a convincing case was argued for removing it.

There are several reasons why framebuffer objects were introduced:

FB1. It can be "expensive" (for some definition of expensive) to validate the framebuffer and all its attached objects. There is a desire to be able to easily recognize that a particular state. combination has been seen and validated previously.

FB2. There is some subset of GL context state which only makes sense in its relationship to the current framebuffer and attached images (red bits, green bits, blue bits, etc, presence or absence of aux buffers or depth buffers, current value of draw buffer(s), read buffer, etc. etc). It would be nice if this state "tracked" changes to the current framebuffer configuration by being part of the framebuffer object state.

FB3. For a while, we considered adding "intrinsic" or "implicit" buffer storage to the framebuffer. This would be used for buffers that were either hidden from the user, like the multisample buffer, or perhaps needed to be explicitly formatted by the driver. If we did have this kind of "intrinsic" storage, then framebuffers would be a lot like textures and would have the same kinds of pressures to minimize vram, sharing storage across objects and contexts as textures did. In fact, they would be similar to cube map texture objects which had 6 attached face images, or mipmaped textures which had a set of mipmap level images. In the end we decided not to use intrinsic buffers, - see issue (13) - but we might decide to add them back in the future. For instance, one option for supporting multisampling is to use an implicit multisample buffer.

FB4. We realized that most of the "hard" issues introduced by this extension were completely orthogonal to the presence or absence of framebuffer objects. All of the same issues apply regardless of whether there is a single non-default framebuffer as part of the context or multiple framebuffer objects. These issues about attaching, (binding) objects, reformatting attached (or bound) images via TexImage/RenderbufferStorage, pixel format combinations, framebuffer completeness, and the relationship between a non-"default" framebuffer and the legacy window sytem framebuffer and pixel format all come in to play either way. So there is actually little implementation or conceptual cost incurred by the introduction of these framebuffer objects.

There were also a few reasons why we considered *not* adding framebuffer objects:

NoFB1. In the absence of "intrinsic" buffers, framebuffer objects only really consist of the attachment state. It is convenient to encapsulate this state into an object, but one could ask if it's any more convenient than say a "blend state" object or a "texture unit attachment state" object, which to date, we have chosen not to add into OpenGL.

NoFB2. As a "state-only" object, there's a question about how much state should be included - at least the attachment state should be included, but what about draw buffers state, what about the viewport state, what about other state? Since drawing the line is hard, we questioned whether we needed these objects.

NoFB3. Some amount of the functionality of the framebuffer objects could be implemented by the application with the appropriate use of display lists.

In weighing (FB1), expense of validating framebuffer state, versus (NoFB1), not wanting to introduce "state only" objects, we realized that framebuffer validation is more expensive than the blend state (for which there is no object in GL) and less expensive than a fragment program (for which there is an object in GL). While it's not exactly clear precisely where on the spectrum of "expense" the framebuffer validation lies, we decided that it may be expensive enough to justify creating a new object type. So we retained framebuffer objects in the API now, with the understanding that if we change our minds it's easier to rip them out later than it is to add them back in later.

(9) Should the routine which allocates a renderbuffer accept an image to initialize the buffer, analogous to how TexImage works?

RESOLUTION: no, it should allocate uninitialized storage

We could have allowed a renderbuffer "image" specification routine, but this would essentially serve the same purpose as a combined "allocate renderbuffer followed by DrawPixels" routine so we decided it was extraneous. The primary purpose of these buffers is to store rendered output anyway, so there was not sufficient demand to support an optimized path for data initialization. See related issue (10).

(10) What should we call the routine that allocates storage for the renderbuffer? This routine would be the moral equivalent of glTexImage.

RESOLUTION: RenderbufferStorage()

```
Options included:
    RenderbufferStorage()
    RenderbufferImage()
    others???
```

This is really a function of how we resolve issue (9).

RenderbufferImage would be appropriate if the allocation routine could take an image to initialize the renderbuffer.

RenderbufferStorage would be more appropriate if the allocation routine does not take an image.

Since the group decided supporting an "initialization" image for a "renderbuffer" was too much overlapping functionality with DrawPixels, RenderbufferStorage was chosen.

(11) The routine(s) which attach a texture to a framebuffer attachment point need to describe which image in the texture they are using, i.e., which cube map face, mipmap level, or 3D texture z-slice/depthoffset/image. Should we have one routine that handles all of these with some arguments ignored for specific texture types/targets? Or should we have a parallel set of routines for 1D/2D/3D, like TexImage does?

RESOLUTION: Option (b) 3 routines for texture, 1 for renderbuffer

Originally, we chose option (b) for reasons of similarity to glTexImage1D/2D/3D. For TexImage2D and FramebufferTexture2D, the texture target was used to select a face on a cube map texture object. Since glTexImage1D/3D used TEXTURE_1D/TEXTURE_3D texture targets, we did the same for FramebufferTexture1D/3D. We also included the texture target in case it was needed for future expandability.

However, some felt uncomfortable with this resolution since it adds 3 framebuffer attachment calls for textures, so we reopened the issue.

Originally we just considered options (a) and (b). We then reconsidered a few additional flavors: (c), (d), and (e)

```
             Options include:

             a) one routine with arguments that are sometimes "ignored"

                For instance <image> is ignored for non-3D textures
                and <face> is ignored for non-cube maps, etc.

                This gives us:

                void FramebufferTexture(enum target, enum attachment,
                                        uint texture,
                                        uint level, enum face, uint image);

             b) routines for 1D/2D/3D, use FramebufferTexture2D for 2D,
                Cube, Rectangle

                Requires use of a texture target to distinguish cube map
                faces on FramebufferTexture2D

                Includes "redundant" texture target for 1D/3D variants
                for consistency and precedent with TexImage1D/3D.

                This gives us:

                void FramebufferTexture1D(enum target, enum attachment,
                                          enum textarget, uint texture,
                                          uint level);
                void FramebufferTexture2D(enum target, enum attachment,
                                          enum textarget, uint texture,
                                          uint level);
                void FramebufferTexture3D(enum target, enum attachment,
                                          enum textarget, uint texture,
                                          uint level, uint image);

            c) same as (b) but add a dedicated routine for Cubemaps

               Question: since we added a Cubemap version, do we need a
               Rectangle variant as well?

                This gives us:

                void FramebufferTexture1D(enum target, enum attachment,
                                          enum textarget, uint texture,
                                          uint level);
                void FramebufferTexture2D(enum target, enum attachment,
                                          enum textarget, uint texture,
                                          uint level);
                void FramebufferTextureCubemap(enum target, enum attachment,
                                          enum textarget, uint texture,
                                          uint level);
                void FramebufferTexture3D(enum target, enum attachment,
                                          enum textarget, uint texture,
                                           uint level, uint image);
```

```
    d) same as (c) but with no texture target parameter

       Question: since we added a Cubemap version, do we need a
       Rectangle variant as well?

        This gives us:

        void FramebufferTexture1D(enum target, enum attachment,
                                  uint texture, uint level);
        void FramebufferTexture2D(enum target, enum attachment,
                                  uint texture, uint level);
        void FramebufferTextureCubemap(enum target, enum attachment,
                                       uint texture, enum face,
                                       uint level);
        void FramebufferTexture3D(enum target, enum attachment,
                                  uint texture, uint level, uint image);


     e) one FramebufferTexture routine with additional arguments
        passed in via another routine.

        There are no "ignored" arguments in this routine.

        The arguments which would be "ignored" by this function
        are passed in as selector state by a separate function.
        These could be specified as a FramebufferParameter
        (implying that they are stored as framebuffer state), or
        as a piece of context state that is copied into the
        framebuffer attachment point at FramebufferTexture time.
        Of the two, context state is much more desirable since
        ARB_render_texture made the mistake of putting the
        selection state in the pbuffer, and this has real
        usability issues for multicontext applications.

        This gives us (two routines)

        void FramebufferTexture(enum target, enum attachment,
                                uint texture, uint level);
        and

        void FramebufferParameter(enum target, enum pname, uint param);
            where pname can be one of
                GL_{attachment}_TEXTURE_CUBEMAP_FACE
                GL_{attachment}_TEXTURE_3D_IMAGE
            and param represents the cube map face or z-slice image.

        Also, option (e) raises 2 questions:

        1. Since the rest of the selection state would come in
           through another function, we have to ask when can
           these selector state variables be changed?

           We had previously decided that we want to pass
           selection state in atomically with the attachment
           request.  To be consistent with this earlier
           decision, this would imply that these variables could
           not be changed dynamically but would be "snapshotted"
```

```
            into the framebuffer attachment point at at
            FramebufferTexture time.  This snapshot could be
            thought of as similar to the way ActiveTexture works.
            This is also similar to the snapshot of the
            transformed raster pos vertex that occurs at
            glRasterPos time.  It is a copy of one piece of state
            into another piece of state, not just a "switch" than
            can be updated later that indicates where other state
            should be stored.

         2. Is the rationale to consolidate FramebufferTexture
            from 3 routines to 1 also a reason to consolidate
            FramebufferTexture and FramebufferRenderbuffer into a
            single attachment routine?  I.e., should there just
            be one routine called FramebufferAttachableImage()?

            If we did this, then we could also move <level> out
            of the argument list and rename the function to,
            perhaps, FramebufferAttach.

              void FramebufferAttach(enum target, enum attachment,
                                     enum objectType, uint name);
              and we'd need to create another enum for
              FramebufferParameter
                  GL_{attachment}_TEXTURE_LEVEL

              or, avoiding the use of verbs in the function
              name, perhaps:

              void FramebufferAttachableImage(enum target, enum attachment,
                                              enum objectType, uint name);

    Rationale:

    (a) was discarded because it was not very extensible in the
    event we need to add additional texture selection state in
    the future (for instance, what if we add TEXTURE_4D
    targets?)

    (c) and (d) were discarded because the introduction of a
    special cubemap routine was undesirable since we were
    considering issue in an attempt to *reduce* the number of
    entry points.  Additionally, (d) was discarded because it
    was felt the texture targets were still required.

    (e) was discarded because the intent was that attachment
    (and the consequent framebuffer validation) was a
    "heavy-weight" operation.  By using a separate routine to
    set part of the attachment state, developers may be
    incorrectly encouraged to assume some attachment state could
    be changed more easily than others.  It was felt it wasn't
    worth this possible misunderstanding just to save some
    function entry points.

    In the end, it was determined that (b) was the lesser of two
    (five?) evils.  (b) also has precedent in the specification
    of texture images via gl{Copy}TexImage.  Finally, (b) is
```

pretty clearly extensible to new attachment routines for future object types.

(12) Do we need a "format group" or "format restriction" API?

RESOLUTION: Yes, but put it in a separate extension for reasons of schedule.

This extension introduces the ability to construct a collection of logical buffers using images of various formats into a framebuffer in a very flexible manner. It is by design more flexible than used to be possible to do by querying for available pixel formats in the window-system glX/WGL/agl API's. As a result, it is possible to construct a framebuffer that is actually not supportable by the implementation and the reasons for the configuration being unsupportable are entirely implementation dependent.

This is why we originally added the CheckFramebufferStatus API. So that the application at least has the ability to determine that a particular, otherwise legal, configuration of framebuffer attachments actually will not work on this implementation.

However, this extension does not provide any very helpful mechanism to find out why things are not supported or what to do to reconfigure the attachments into a supported configuration.

This is a very difficult problem to solve. glX/WGL/agl solved this problem by allowing the application to specify a request for a configuration and letting implementation provide a "best match". Additionally, glX and WGL also allow for the enumeration of all possible supported configurations.

Various schemes like these were considered but they were all quite complicated (possibly as complicated as the windowing system API's we are trying to replace). Consequently, we decided to investigate some additional approaches.

One of these approaches is to specify "allocation and usage" hints prior to the routines which allocate buffers (TexImage/RenderbufferStorage) that will somehow indicate an intended configuration and then let the implementation use this additional information when selecting internal formats for textures and renderbuffers. The GL already has the freedom to pick any internal format it wants for textures and renderbuffers (subject to invariance requirements), and so we would like to leverage this freedom and influence the choice with an additional channel of information.

One example, though not the only one, is some API to let the application specify it would like to be able to use a color buffer, depth, and stencil buffer. The implementation would take advantage of this information when allocating textures and renderbuffers and only choose internal formats for color, depth, and stencil textures and renderbuffers that could be guaranteed to be used together. For instance, the user could call:

```
    FormatRestriction(GL_COLOR | GL_DEPTH | STENCIL);
```

or perhaps

```
    FormatRestriction(GL_32_BITS_COLOR_DEPTH_STENCIL);
```

and then when the user called TexImage with a color buffer, the GL would only pick color formats that could definitely be used with depth and stencil buffers. The effect of this API would be to "restrict" the avaible choices to the GL to the subset of compatible formats. In this way, the possibility of encountering an implementation-dependent reason for failing "framebuffer completeness" would be greatly reduced or perhaps entirely eliminated.

In any event, specifying this "FormatRestriction" API was going to take additional time and we wished to get this base EXT_framebuffer_object specification done and shipping as soon as possible. So we agreed to defer this "format restriction" API specification to a later extension, with the intent to develop this API or some other solution to this problem as soon as possible.

(13) Do we need intrinsic buffers in addition to renderbuffers?

RESOLUTION: no

When intrinsic buffers were initially proposed, the format and dimensions of an intrinsic buffer could mutate in order to provide compatibility with the other images attached to a framebuffer object. After much debate and a series of votes, intrinsic buffers had lost both of those properties. (See issue 36.) In the end the working group decided that the crippled form of intrinsic buffers do not provide enough added value to justify their existence.

(14) Is it necessary to require that all the logical buffers of a framebuffer object have the same dimensions?

RESOLUTION: Yes. Matching dimensions are required for simplicity. If the dimensions do not match, the framebuffer object will not be "framebuffer complete".

It could be useful to use a single large depth buffer when rendering to many textures of several different sizes. This is something that could be added later by a layered extension that relaxes the matching dimension restriction. Supporting heterogeneous sized logical buffers requires defining where in a larger buffer the smaller results are written, and deciding what guarantees can be made and what should be left undefined.

(15) *What happens when TexImage or CopyTexImage is called on a texture image that is attached as an image of the currently bound framebuffer object?*

RESOLUTION: resolved, {Copy}TexImage will redefine the texture image, which can affect the completeness of the framebuffer to which it is attached, and possibly cause the currently bound framebuffer to start failing the framebuffer completeness test.

As far as {Copy}TexImage (or RenderbufferStorage) are concerned, there is nothing "special" about a texture image (or renderbuffer) attached to a framebuffer object. Attempts to redefine attached images in this manner should succeed. However, if the redefined image is no longer appropriate for the relevant attachment point in the framebuffer it is attached to, then it's possible the framebuffer may start failing the framebuffer completeness test.

Another option that was considered involved having TexImage and CopyTexImage result in INVALID_OPERATION and do nothing when the target texture is bound for render-to-texture. This idea was rejected because, in the multicontext case, one context could change the attachments of a shared framebuffer and cause another context to suddenly start generating errors on {Copy}TexImage calls. This extension has tried to avoid introducing asynchronous generation of gl errors.

Still another option that was considered was "orphaning" the old texture memory such that it could still be used as a framebuffer attachment but the texture would get newly allocated storage. However, this implied a side-ways copy of the texture object memory or the image for its continued use as a framebuffer-attachable image, and was therefore rejected.

For the purposes of comparison, consider that ARB_render_texture faced a similar question and resolved it by implicitly unbinding the texture from the pbuffer when TexImage is called.

(16) *What happens when TexImage or CopyTexImage is called on a texture object that is attached as an image of a framebuffer object that is not bound to the current context?*

RESOLUTION: resolved, {Copy}TexImage will redefine the texture image, which can affect the completeness of the framebuffer object to which it is attached. When the framebuffer object is bound to the context, it may start failing the framebuffer completeness test. If the framebuffer object is bound in another context at the time {Copy}TexImage is called, then the framebuffer object may start failing the framebuffer completeness test in the other context.

The rationale for this decision is the same as for issue (15).

However, since in this case the relevant framebuffer is not current, there is no guarantee that this framebuffer revalidation or invalidation will happen until the next time the framebuffer is bound to a context.

The texture (or renderbuffer) state is changed immediately, regardless of whether the texture image (or renderbuffer) is attached to a framebuffer object. However, a context other than the one issuing the {Copy}TexImage operation might not notice the state change until after it has (re)bound the framebuffer object or reattached the texture image.

This is intended to be similar to what happens in the multicontext case when the state of a shared texture object is changed by another context. There is no guarantee that texture state change will be visible in the current context until the current context binds the texture object again.

(17) Why is render to vertex array missing?

RESOLUTION: Render to vertex array is separate functionality from render to logical buffer or render to texture. RTVA can be added as a separate extension. The framework is general enough to support more than one way of adding RTVA, without deciding today on the details of a particular RTVA implementation.

One idea is to define a way to interpret a vertex array or buffer object, which is inherently byte-oriented linear, as a framebuffer, which is inherently component-oriented and dimensioned, and then call FramebufferArrayEXT like this:

```
FramebufferArrayEXT(FRAMEBUFFER_EXT, COLOR, buffer_obj);
```

Another idea is to define a general way to interpret a component-oriented dimensioned image, such as a texture or a color buffer, as a byte-oriented vertex stream. Using this approach one would render vertex attributes to a renderbuffer, to a texture image, or to an AUX buffer, and then use the image data directly as a vertex array.

There is controversy over which RTVA method(s) should be supported. One goal of EXT_framebuffer_object is to ship render-to-texture and render-to-logical-buffer functionality today while leaving the door open to add one or more RTVA solutions in the future.

(18) What function should perform the action of attaching a texture image to a framebuffer for rendering purposes?

RESOLUTION: The new FramebufferTexture*EXT functions perform this action.

Options that were considered include overloading BindTexture, using a FramebufferParameter function, and adding a new function.

BindTexture is problematic because it creates a new texture object with default state if the name is previously unused, but the default state has no dimensions, dimensionality, or format.

One reason that FramebufferTexture*EXT was well-received is because it sets, in one atomic operation, all framebuffer attachment state for both texture image and renderbuffer type of attachments. Given the polymorphic nature of framebuffer-attachable images, this guarantees that all framebuffer attachment state is in a consistent configuration, without having to define confusing precedent rules between competing (texture image and renderbuffer) pieces of framebuffer attachment state, or having to create enables (either a tri-state enable or separate enables again with precedence) to select texture image or renderbuffer attachment state as the "active" set of state.

This decision also makes it simpler to specify how a framebuffer-attachable image is detached from a framebuffer--it would be confusing if detaching a texture image resulted in *attaching* a renderbuffer simply because texture image attachment state takes precedence over renderbuffer image attachment state.

(19) What should happen if the texture argument given to FramebufferTextureEXT is an unused texture name? And similarly, what should happen if the renderbuffer argument given to FramebufferRenderbufferEXT is an unused renderbuffer name?

RESOLUTION: resolved, (a) this is an error.

Options included:

a) throw an error at Framebuffer{Texture|Renderbuffer}

b) texture/renderbuffer is created just like Bind{Texture|Renderbuffer}

c) no error, but the framebuffer cannot be "framebuffer complete" until a texture/renderbuffer by that name has been created and satisfies the rules of framebuffer completeness.

This is interesting because on the one hand we might like to adopt the model that we simply catch all the invalid state combinations when determining framebuffer completeness, i.e., option (c). This has a certain consistency but then what does it mean to call FramebufferTexture{1D|2D|3D} when the target of the texture name is not yet known? How should the other arguments to those calls be validated?

Option (b) was rejected as it would introduce a second way to create a texture/renderbuffer object. I.e., both

BindTexture and FramebufferTexture would create the texture object.

Since there are "target aware" FramebufferTexture{1D|2D|3D} calls, the app already has to know the target prior to calling FramebufferTexture. Also, the texture target of a given object is immutable once set. An app can not set it and then change it later so this is really just an issue with the order in which they call the relevant functions. Consequently, requiring that the user call BindTexture prior to calling FramebufferTexture does not seem to be a burden. So this should be an error, since it's probably a mistake on the user's part in the first place.

(20) What should happen if the texture argument given to FramebufferTextureEXT is the name of an existing texture object, but the texture has no texture image (i.e., TexImage has never been called)? Similarly what should happen if the renderbuffer argument given to FramebufferRenderbufferEXT is the name of an existing renderbuffer, but the named renderbuffer has no storage (i.e., RenderbufferStorage has never been called?)

RESOLUTION: resolved, option (c) - no error, but the framebuffer object cannot be "framebuffer complete" until the state of the texture image satisfies the rules of framebuffer completeness.

Same options as issue (19), these include:

a) throw an error at Framebuffer{Texture|Renderbuffer}

b) texture/renderbuffer is created just like Bind{Texture|Renderbuffer}

c) no error, but the framebuffer cannot be "framebuffer complete" until the texture image or renderbuffer satisfies the rules of framebuffer completeness.

This is an issue because you could be attempting to attach a texture (or renderbuffer) to a framebuffer attachment point prior to the application having called TexImage (or RenderbufferStorage) to define the width/height/format of the framebuffer-attachable image.

At first, this seems similar to issue (19), so we could throw an error in this case too. It is different for two reasons however. First, there are default values for the texture object and renderbuffer object state. Second, the values of the width/height/format/etc for the texture object are mutable, unlike the texture target of the texture object. There is really no difference between the case where GL uses the default values for an object, and the case where the user explictly set the state equivalent to the default values using TexImage (or RenderbufferStorage). Because this state is mutable, it must be tested anyway when framebuffer completeness is determined.

Therefore, we simply defer the check for whether the texture/renderbuffer state is appropriate for the framebuffer attachment point until determination of framebuffer completeness. If the state is not valid, then the framebuffer will not be complete, regardless of whether or not TexImage/RenderbufferStorage has been used to create storage for the texture level (renderbuffer).

(21) *What happens when DeleteTextures is called on a texture that is attached to a framebuffer object? Similarly, what happens when DeleteRenderbuffers is called on a renderbuffer that is attached to a framebuffer object?*

RESOLUTION: resolved, see issue (77)

(22) *How do you detach a texture or renderbuffer from a framebuffer object? Should we use two routines or create a detach routine?*

RESOLUTION: resolved, 2 routines

If the user calls either FramebufferTexture with a zero texture name, or FramebufferRenderbuffer with a zero renderbuffer name, then the it as if nothing is attached to the specified attachment point.

There was a concern that having two routines be able to set the framebuffer attachment state to "none" was confusing. However, the idea is simply that for any object that can be attached to a framebuffer, there should be a routine that can set up the attachment and return the framebuffer to the default "nothing attached" state.

The implication here is that the default state for framebuffer attachments is:

```
    attachment object type = GL_NONE, and
    attached object name   = 0
```

(23) *Should it be legal for the framebuffer state to pass through invalid configurations? (I.e., depth and color buffer sizes don't match, etc)*

RESOLUTION: resolved, "yes"

It's easier for the application if the render target state is allowed to pass through invalid configurations when transitioning between two valid configurations. A consistency check is defined to determine if a configuration is valid.

As long as everything is valid at render time, transient invalid states are allowed.

(24) What happens when you try to draw to a framebuffer that is not "framebuffer complete"?

RESOLUTION: resolved, rendering is disabled, and an error is generated. See issue (64) as this issue is essentially a duplicate of that one.

(25) What should happen on a query of framebuffer state while the framebuffer is invalid? For instance, what does a query of RED_BITS return if the currently bound framebuffer is not "framebuffer complete"?

RESOLUTION: resolved, there's no issue here. Attempts to query bit depths should return the "real" answers.

For instance, if there's no color buffer attached to the framebuffer attachment point, then attempts to return RED_BITS could return zero. If there is a color-renderable image attached, then RED_BITS would return whatever the RED_BITS are, regardless of the valid/invalid state of the framebuffer.

Other options include returning some kind of magic value or generating an error if the framebuffer is invalid. However, any "magic value" would simply be a duplicated query for the framebuffer completeness status. Also, returning an error would be problematic because another context can make a framebuffer invalid and we have been trying to avoid any API in which one context can cause another context to start generating errors asynchronously.

(26) What happens when you try to read (e.g. ReadPixels) from a framebuffer that is not "framebuffer complete"? Reads cannot be "disabled" or "ignored" in the same way that rendering can.

RESOLUTION: resolved, generate a GL error. See issue (65).

Originally this was resolved as "undefined pixels are generated, but no error"

Initially, generating an error was rejected for a few reasons. First, it is asymmetric with the behavior for drawing - when the framebuffer is not complete, drawing is disabled. We would like to be consistent here. Second, there are no other cases where ReadPixels or CopyTex{Sub}Image will generate an error based on the state of the framebuffer and we didn't want to introduce one. Third, there is already a pixel ownership requirement in order to get defined results back from reading the framebuffer, so if we simply behave as if incomplete framebuffer fails ths pixel ownership test, then we can leverage that already specified behavior for reading the framebuffer.

For these reasons, we initially choose to have reads from an incomplete framebuffer return undefined pixel values and not generate a GL error.

However, once we subsequntly resolved issue (64) to say that rendering with an incomplete framebuffer generates an error, we decided again for reasons of symmetry that reading from an incomplete framebuffer should also generate an error. (And most likely the same error.)

So in the end, we decided that reads (e.g., ReadPixels and CopyTex{Sub}Image) in this case would result in an error to be named in issue (65).

See also related issue (73), describing ReadPixels of color data from a complete framebuffer while READ_BUFFER is NONE.

(27) What happens when you query the number of bits per channel (e.g., DEPTH_BITS) prior to the consistency check being run when intrinsic buffers are in use, since implementations are allowed to select a number of bits for an intrinsic buffer at consistency check time to give a better chance of a consistent state being reached?

RESOLUTION: This is not an issue since we don't have intrinsic buffers, see issue (13). We are keeping this issue in the issues list just in case we ever go back and add something like this to a future API.

If we would have retained the intrinsic buffer api (i.e., glFramebufferStorage) or if some future API adds it back in, then one possible resolution of this problem would have been to simply say that a query of the number of bits prior to the consistency check being run will produce an answer that is subject to change.

This is preferable to some other possible resolutions that have been discussed (e.g., having the query cause a validation to occur implicitly, thereby "baking" in the answer) because it is the one least likely to introduce unexpected side-effects to an operation as seemingly innocuous as a query.

A possible variant of this proposed resolution would have been to have the query return a number of bits that is guaranteed to be less than or equal to the actual number of bits that will eventually be used. This may or may not be a useful guarantee. We could have also had the query return 0 or -1 as a signal that the framebuffer is incomplete.

Again, this is all moot since we decided against this style of intrinsic buffers in this extension.

(28) What should the <image> parameter to FramebufferTexture3DEXT actually be called?

RESOLUTION: resolved, "zoffset"

This parameter could have been called <image> or <slice>. <depth> or <zoffset> might also be appropriate. The reason the answer here is non-obvious is that normally 3D textures are specified all at once, not one 2D "slice" at a time (TexImage3D takes one big array that represents all three dimensions at once, for example), and because texture coordinates for TEXTURE_3D targets are normalized floating-point numbers, just as they are with TEXTURE_2D targets, not integer indices.

The GL uses the term "image" to mean "slice" in a few instances. For example, pixel unpack parameters UNPACK_SKIP_IMAGE and UNPACK_IMAGE_HEIGHT describe state related to the "slices" a 3d texture.

However, in some ways the act of rendering into a texture is most similar to CopyTexSubImage3D, which also redefines a texture's contents (but never its format or dimensions) based on the contents of the framebuffer. The "zoffset" parameter to CopyTexSubImage selects a particular 2D image (depth "slice") of a 3-dimensional texture. "zoffset" is a coordinate, and the parameter to FramebufferTexture3DEXT is also a coordinate. "Image" typically refers to an array of pixels.

We already use the term "image" throughout this extension to talk about 2d arrays of pixels beyond their use in 3D textures. It is a little confusing to overload "image" to also mean Z coordinate in FramebufferTexture3DEXT.

For the sum of these reasons, we decided "zoffset" is a better name than "image", for the parameter to FramebufferTexture3DEXT.

(29) Should GenerateMipmap functionality be included in this extension or put in it's own extension?

RESOLUTION: resolved, yes, include this functionality

It is arguably useful separately, i.e., without all this machinery. However, it's also kind of required here to have some kind of way to deal with the interaction with SGIS_generate_mipmap. Probably we should just include it here. (maybe also a separate extension?)

It's easier to define when automatic mipmap generation happens for a traditional non-rendered texture than it is for a texture that is modified by rendering-to-texture. If GENERATE_MIPMAP were to cause a rendered-texture's mipmaps to be automatically generated, presumably generation would occur when either the texture is detached from the framebuffer or when the framebuffer is unbound. If neither of these events occur, should automatic mipmap generation also occur when the texture is bound to a texture unit (of same or different context?)

It's believed the recommended way of achieving maximum performance using this extension is to make all attachments during initialization, and then not change attachments in the steady state. This reasoning is, after all, a major reason for introducing framebuffer objects. If an application does not detach textures from framebuffers, then what event triggers mipmap generation? An explicit GenerateMipmap works well here.

Would the base level have to actually be modified in order for mipmap generation to occur? How should "modified" be defined?

If the application rendered to each level of the texture before detaching the texture or unbinding the framebuffer, would automatic mipmap generation happen anyway? (This implies the application needs to set GENERATE_MIPMAP to FALSE before rendering to the texture, but maybe that's OK.)

Historical background: One reason for introducing GenerateMipmap in the context of the original uber_buffers proposal was that uber_buffers lacked a Begin-time consistency check, but instead prevented the framebuffer from ever getting into an inconsistent state (once validated). Operations such as TexImage that can change the dimensions and format of a tetxture's levels were disallowed when the texture was attached to a framebuffer. Since automatic mipmap generation can change the dimensions and format of a texture's levels, that meant that automatic mipmap generation could not be performed in some cases, but there was no good way to communicate this error to the application. Hence there really was a need for a separate GenerateMipmaps function. This restriction does not apply to the current API because the semantics of an incomplete framebuffer are different now. Nevertheless, we decided to retain this manual mipmap generation as part of this extension.

(30) Do the calls to deal with renderbuffers need a target parameter? It seems unlikely this will be used for anything.

RESOLUTION: resolved, yes

Whether we call it a "target" or not, there is *some* piece of state in the context to hold the current renderbuffer binding. This is required so that we can call routines like RenderbufferStorage and {Get}RenderbufferParameter() without passing in an object name. It is also possible we may decide to use the renderbuffer target parameter to distinguish between multisample and non multisample buffers. Given those reasons, the precedent of texture objects, and the possibility we may come up with some other renderbuffer target types in the future, it seems prudent and not all that costly to just include the target type now.

(31) What should happen if you call FramebufferTexture{1D|2D|3D} with a texture name of zero?

RESOLUTION: This will detach the image from the specified attachment point in the currently bound framebuffer object.

For reference, this reason this is problematic because there is not really a "texture object zero"

Texture name zero does not define an object but defines context state (one texture named zero, per target, per context). The textures referred to by the name zero are never shared across contexts. So the behavior of framebuffer objects shared by multiple contexts where each is attached to the context's texture named zero seems odd at best, and confusing at worst. As such, it was decided to not allow a framebuffer to attach to texture named zero.

Another option would have been to make this an error. If we had done this, then we would need a specific function to detach a texture from an attachment point. That is, we would have needed to create something like a dedicated DetachFramebufferAttachableImage() entry point.

(32) Should there be a renderbuffer object with the name of zero?

RESOLUTION: NO.

By way of symmetry with textures, renderbuffer zero, if it existed, would not be an object. It would be a non-shareable piece of the context state. There would be one renderbuffer named zero per target per context.

If we can't share renderbuffer name zero, then also by way of symmetry with textures, we would not want to support attaching renderbuffer name zero to a framebuffer.

So, if it can't be used as a rendering destination, then a renderbuffer name zero would seem to serve no purpose as a state container.

However, we'd like to retain the use of name zero in certain routines with special semantics, particularly for detaching non-zero renderbuffer objects from the framebuffer and context. See issue (33).

On implication of this decision is that state setting/getting routines that operate on the currently bound renderbuffer should throw a GL error if no renderbuffer is bound/attached. A similar choice was made in the ARB_vertex_buffer_objects specification which also had special semantics for object zero.

Also note, another option considered was making object zero a full fledged, shareable object just like the non-zero object names. This was rejected as being too different from texture/program vbo's/etc., possibly leading to confusion.

(33) What should happen if you call FramebufferRenderbuffer or BindRenderbuffer with a renderbuffer name of zero?

RESOLUTION: This will detach the image from the specified attachment point in the currently bound framebuffer object.

This is resolved exactly the same way as issue (31) was resolved for textures, and for the same reasons.

Similarly, calling BindRenderbuffer with a name of zero will unbind the currently bound renderbuffer from the context.

(34) Should there be a way to query a framebuffer object for its attached texture and/or renderbuffers? If so, how, and what should be the query result when attached textures or renderbuffers have been deleted?

RESOLUTION: resolved, yes

In general, OpenGL lets you query settable state, so we allow this.
To see what this query should look like, see related issue (51)

This issue also raises the question about what values should be returned for attached objects if the named objects have since been deleted. This can happen if the textures were attached to non-currently bound framebuffers or attached to framebuffers in other contexts. Three possible solutions include:

a) Don't support this query.

b) Return zero if no texture has ever been attached. Return zero if the attached texture has been deleted.

c) Return zero if no texture has ever been attached. Return the name of the texture that was attached even though it has been deleted.

Option (a) was rejected as we would like settable state to be queriable.

So, for this extension originally we choose option (c). However, we have since decided, in issue (21), that DeleteTexture and DeleteRenderbuffer will first detach the texture/renderbuffer from any attached framebuffer objects *in this context*. In principle, the application can't tell the difference between the texture getting deleted now or later, so whether the texture is actually detached from the current framebuffer now and other framebuffers when they are bound, or the texture is actually detached from all framebuffers at once is moot. In practice, this means that options (b) and (c) are essentially indistinguishable for a single context case.

However, it's worth noting that if the texture is deleted and attached to a framebuffer which is current in another context, the standard rules about undefined behavior of state modifcations of shared objects in other contexts will still applye.

This means that the texture may or may not be detached (and thus deleted) from that other context's current framebuffer until the next BindFramebuffer (or FramebufferTexture/FramebufferRenderbuffer?) in the other context.

(35) Earlier proposals included a way to create some memory and then attach it to a texture object. Should this extension include this feature?

RESOLUTION: no.

This was considered when this extension was intended to be a more general purpose memory manager. Since this extension has been retasked to focus in on render-to-X functionality, this feature was not necessary.

(36) Earlier proposals had renderable memory constructs which could change internal format or dimensions to meet intra-framebuffer compatibiltiy requirements of individual vendors' hardware platforms. Should this extension have these kind of malleable format objects?

RESOLUTION: no.

Such malleability leads to invariance problems when formats change. For example, if bits per pixel is decreased then increased back to the original value, some precision is lost.

Some IHVs wanted to require format conversion of existing contents in all cases where the format changes. This sort of invariance would be an acceptable side-effect. The suggestion was to think of the action of rendering to a texture as an extended non-atomic TexImage call. TexImage is allowed to change the format of an existing texture image. It was claimed that such intrinsic buffers are more convenient in many applicaitons than are the explicitly managed renderbuffers.

Other IHVs expressed a strong opinion against implicit format conversions, but instead wanted to invalidate the buffer's contents whenever the format changed. It was difficult to define the set of operations that might cause the format to change, so it was difficult to define when the contents could become invalidated. If the contents were invalidated by a format change, the API under consideration made it cumbersome for the application to detect and handle this condition. In the end, under the buffer content invalidation approach, application code would not be any better off than if the appliation instead used the explicit renderbuffers. For the type of intrinsic buffers that could not change format and dimensions dynamically, the claim that intrinsic buffers were more convenient than renderbuffers was no longer true.

The working group voted for the latter: no implicit format changes. Instead the format would be immutable once it is known.

A secondary issue is the question: are the buffer contents invalidated when the dimensions change, are the contents scaled, or are the contents are clipped/padded (with some sort of gravity). This issue could be avoided by requiring explicit, rather than implicit, resize of intrinsic buffers.

The working group voted for no implicit change in the dimensions of intrinsic buffers, and finally for the removal of intrinsic buffers altogether.

(37) In order to abstract hardware dependent compatibility requirements, this API introduces a function called CheckFramebufferStatus to check for compatibility prior to rendering. CheckFramebufferStatus returns a value which indicates whether or not the framebuffer object is "framebuffer complete", and framebuffer completeness depends in part on hardware dependent constraints. The hardware dependent aspect represents a new concept in OpenGL. Therefore, should an app be required to call this function to help "enforce" the notion that apps should be on the lookout for failure?

RESOLUTION: no. Calling CheckFramebufferStatus is not required.

The group considered requiring a call to CheckFramebufferStatus after changing framebuffer state or attachment points in order to "enable" rendering. It was hoped that requiring a call to CheckFramebufferStatus would push developers to write code which is more platform independent. Ultimately though, since the API can't require applications to actually observe and deal with a validation failure, that it was not worth it to make this function call required. There was also feedback from some developers that requiring this call would be cumbersome and undesirable.

Note, however, that the framebuffer is effectively validated implicitly at every rendering (and reading) entry point. These include glBegin, gl{Multi}Draw{Arrays|Elements}, gl{Draw|Copy|Read}Pixels, glCopyPixels, glReadPixels, glCopyTex{Sub}Image, etc.

Applications are strongly advised to test framebuffer completeness with CheckFramebufferStatus after setting up or changing the configuration of a framebuffer object, and to handle the possible failure cases with a fallback plan that selects a different set of internal formats of attached images. See usage example 6. Section 4.4.4.2 lists the operations that can cause the framebuffer's status to change.

In addition, a "format group" API, has been proposed as a means of programmatically determining a set of internal formats that are guaranteed to be compatible with respect to framebuffer completeness. This API would be specified in a layered extension as suggested in issue (12)

(38) Do we need to support multiple render targets, i.e., ARB_draw_buffers?

RESOLUTION: Yes.

ARB_draw_buffers is going to be part of OpenGL 2.0 so we'd better support it.

(39) How should we support ARB_draw_buffers?

```
RESOLUTION: refactored into the following issues:
            (53), (54), (55), (56), and (57)
```

(40) (How) should we support accum buffers?

RESOLUTION: defer this until (shortly) after this extension.

Accum buffers appears to be very simple to specify and implement. Basically, we would need to add a new internal ACCUM format that can be passed to RenderbufferStorage. We would also need to add an ACCUM attachment point in the framebuffer that could be used to point to one of these ACCUM format renderbuffers. A new ACCUM format is needed because the ACCUM buffer is defined by GL to be signed floating point value, unlike other internal formats.

Also note, the above solution is the exact same one we are using for STENCIL buffers as well (i.e., an internal format enum and an attachment point).

We could also decide if this new ACCUM internal format can be used with textures in addition to renderbuffers, for creating images that can be attached to the accum buffer attachment point.

Supporting accum was deferred for this extension, primarily for time-to-market reasons, and as it was not critical for most render-to-texture applications. However, we intend to work on some kind of "EXT_accum_renderbuffer" extension shortly.

Since this was deferred, we need to define what happens when you call the various Accum operations on a non-default framebuffer object. We considered adding spec language that would generate an error on Accum operations. However, it seems like we can simply leverage whatever legacy behavior is currently defined for when the pixel format has no accum buffer. This is the case in this extension as we have defined no way to attach or enable an accum buffer. Chapter 4 on page 188 already says that "If there is no accumulation buffer, or if the GL is in color index mode, Accum generates the error INVALID OPERATION", so we don't actually need any additional language of our own.

(41) (How) should we support multisample buffers?

RESOLUTION: defer this until (shortly) after this extension.

Supporting multisample was deferred for this extension, primarily for time-to-market reasons and because it's not entirely clear what is the "best" API for exposing multisample. However, we intend to work on some kind of "EXT_multisample_renderbuffer" extension shortly.

Since this feature was deferred, we need to define what happens when you try to enable multisample on a non-default framebuffer object. For now we need some way to *not* do multisampling. This can either be that we set SAMPLES 1 and SAMPLE_BUFFERS to 0, or we say that Enable/Disable(MULTISAMPLE) is ignored. This is actually related to issue (62) - should SAMPLE_BUFFERS change when using a non-default framebuffer or when attachments change? When/if we define and export "EXT_multisample_renderbuffer" extension, this state will again have significance.

A discussion of how we might support this feature follows:

There are several considerations here: First, we'd like something simple to specify, implement, and use. Second, we'd like to not delay this extension's approval, implementation or adoption for this particular feature. Third, we are trying to replace pbuffer functionality, which does support multisampling (at least in principle), so we'd like to not take a step backward in functionality if possible.

However, this extension is *not* trying to "improve" the traditional multisample support. If we do anything, we will simply expose the existing multisample buffer semantics without causing undue implementation burden.

Finally, if an implementation is currently taking short-cuts to GL's traditional "per-pixel-resolve" multisample semantics, we'd like for this extension to continue to allow the exact same short-cuts (to whatever extent the core GL spec does or does not allow those short-cuts). If someone later decides to go an revamp multisampling support in general, they can update this extension at the same time.

Given the above, it appears that the options include:

A) Don't support it. In other words, you can't use mulitsampling and EXT_framebuffer_object. The multisample state is either ignored, or causes the framebuffer to not be complete, or generates some kind

```
            of error.

         B) Create a separate multisample renderbuffer that can be
            attached to a new framebuffer attachment point.

            The reason that we might need a separate
            RESOLUTION_BUFFER is that all renderable color buffer
            formats might not be usable for multisampling on all
            implementations.

            Also, this option would allow multiple framebuffers to
            share the storage for multisample buffers under the
            control of the application.

            Depth sample buffers and stencil sample buffers
            wouldn't necessarily need resolution buffers, but that
            could be added by some future extension.

            This option has several variants:

              B1) Create MULTISAMPLE and/or RESOLUTION_BUFFER
                  internal formats for renderbuffer objects that
                  can be used with RenderbufferStorage.  The
                  samples buffer and the resolution buffer would be
                  allocated and attached to the framebuffer
                  separately.  Having them be separate allows the
                  samples to be deleted after rendering if desired.

                  One issue with this option is that somehow you'd
                  need to specify the number of samples maybe using
                  glFramebufferParameter or
                  glRenderbufferParameter.

              B2) Perhaps, instead of using a single internal
                  format called MULTISAMPLE, use a set of internal
                  formats like MULTISAMPLE_1_SAMPLE,
                  MULTISAMPLE_2_SAMPLE, MULTISAMPLE_4_SAMPLE, etc.
                  This is problematic for supporting depth/stencil
                  multisampling unless we want an explosion of
                  color/depth/stencil multisample internal formats.
                  It's also problematic if MRT draw buffers need to
                  be multisampled because we'd need a number of
                  enums able to support 1 to N draw buffers times
                  the number of sample patterns we support.

              B3) Have RenderbufferStorage always take a number of
                  samples.  We could do this if option (B2) is
                  insufficient due to the need to support DEPTH or
                  STENCIL multisampling, which we probably will.
                  We would then allow the internal format to choose
                  DEPTH, STENCIL, or RGBA/etc.  This is clean but
                  it means that the user would always need to
                  specify a number of samples even when the value
                  is "1".

              B4) Pass in a variable length argument list to the
                  renderbuffer allocation routine, and some of the
```

```
                arguments would indicate intended usage
                (COLOR/DEPTH/MULTISAMPLE) others would indicate
                internal format (RGBA/DEPTH24) and others would
                indicate number of samples.  This is how
                EXT_compromise_buffers dealt with this problem,
                though people didn't seem to like this variable
                length argument list.  EXT_render_target didn't
                deal with this problem so doesn't offer any
                guidance here.

            B5) Create a new RENDERBUFFER_MULTISAMPLE
                renderbuffer target type and a corresponding
                allocation routine, perhaps called
                RenderbufferMultisampleStorage().  This is
                analogous to how textures have their own
                allocation routine per target type
                (TexImage1D/2D/3D, etc).

                With this option, we could preclude
                non-multisample targets from being attached to
                non-multisample attachment points as well.

            B6-B10) Any of the above options can be implemented
                    with either a single monolithic mulitsample
                    buffer that contains the samples for all draw
                    buffers, depth and stencil and a single
                    attachment point, *OR* with independent
                    multisample buffers for each draw buffer and
                    depth and stencil and independent attachment
                    points for each.

        C) Use some kind of "behind the scenes" mulitsample buffer.

           This option also has several variants:

            C1) An "implicit" multisample buffer that is simply a
                property of the framebuffer object.  Each
                framebuffer object could have its own multisample
                buffer(s).  Multisampling would be enabled with
                some kind of FramebufferParameter call.  This
                implies that each framebuffer has memory
                allocated with it.  It further implies that the
                contents of the multisample buffer are
                framebuffer state and are thus retained with the
                framebuffer object.

            C2) We don't say anything except that we say the
                value of the glEnable(MULTISAMPLE) is still
                respected and we render as directed.  This is
                similar to (C1) but we don't go so far as to say
                that the multisample buffer(s) is/are retained
                per framebuffer object.  In other words, a call
                to BindFramebuffer() and changes to framebuffer
                attachments may or may not retain multisample
                buffer contents.  Valid implmentations of this
                would include a multisample buffer per
                framebuffer or one per context.
```

```
          D) something else (hopefully simpler?)

    (42) What set of framebuffer targets should the initial extension
         support?

            RESOLUTION: resolved, (D) single target

            Basic possibilities include:

            (A) DRAW_AND_READ_FRAMEBUFFER_EXT

            (B) DRAW_FRAMEBUFFER_EXT
                READ_FRAMEBUFFER_EXT

            (C) DRAW_FRAMEBUFFER_EXT
                READ_FRAMEBUFFER_EXT
                DRAW_AND_READ_FRAMEBUFFER_EXT

            (D) FRAMEBUFFER_EXT

            The fundamental question is: must framebuffer binding points
            mimic the expressiveness of the window-system function
            MakeContextCurrent, which is described in the glX spec and
            the ARB_make_current_read extension?

            It was not immediately clear how to specify the distinction
            between a READ and a DRAW framebuffer in the context of the
            existing read/draw buffer semantics, given that this
            extension relaxes the "compatibility" requirement between
            read and draw drawables.  How would the value of RED_BITS
            for the read framebuffer be queried if it is different than
            the value of RED_BITS for the draw framebuffer?  What
            exactly is the set of implementation dependent state (see
            the "Implementation Dependent *" state tables in chapter 6)
            that can differ between read and draw framebuffer objects?

            When using MakeContextCurrent, the context's and drawable's
            FBconfig (or pixel format) must be "compatible" or else the
            results are implementation dependent.  But
            EXT_framebuffer_object cannot afford to swing such a large
            "undefined" stick, because it is more likely that
            framebuffer objects are incompatible in this sense, and
            because the "pixel format compatility" of a framebuffer
            object is dynamic--by changing attachments or redefining the
            internal format of an attached texture image.

            The value added by ARB_make_current_read through
            MakeContextCurrent is less relevant to
            EXT_framebuffer_object.  EXT_framebuffer_object enables
            rendering to a texture, and textures are objects with a
            clearly defined mechanism for use as the source of a pixel
            copy: rather than using CopyPixels to move pixels from the
            READ_BUFFER to the DRAW_BUFFER(s), an application can simply
            use the source data as a texture and then draw a
            screen-aligned textured quad to the framebuffer.
```

Additionally, adding separate DRAW and READ bindings in the future is pretty straightforward. One solution would be to say that FRAMEBUFFER_EXT is the DRAW framebuffer, and name the new READ framebuffer FRAMEBUFFER_READ_EXT. Add a new BindFramebuffer-like function which takes two framebuffer names--one for DRAW and one for READ. The current BindFramebuffer function binds a single object to both FRAMEBUFFER_EXT and FRAMEBUFFER_READ_EXT.

So, we defer the additional targets until need has been proven, and go with the simpler option (D) for now.

(43) *In order for a framebuffer object to be "framebuffer complete", must all textures attached to the framebuffer be mipmap complete (or mipmap cube complete if cubemap texture)?*

RESOLUTION: resolved, no

The reason this is a consideration is that some architectures require framebuffer-attachable images to be located in graphics memory when rendered to, and it may be more convenient to allocate and store a texture in graphics memory only if the texture is mipmap (cube) complete--i.e., the size and format of all levels are consistent in the normal sense of texture compeleteness.

However, since framebuffer attachment points only really deal with single images of a texture level, it seems excessive to require the state of the other levels of a texture to affect the validty of the framebuffer object itself.

Addtionally, the same difficulties around "incomplete" textures already apply to traditional CopyTexSubImage, and we have been trying to make the render-to-texture semantics similar to CopyTexSubImage.

Therefore, we chose not to treat render to texture any differently than CopyTexSubImage and do not require that the attached texture is mipmap (cube) complete.

(44) *What should happen if a texture that is currently bound to the context is also used as an image attached to the currently bound framebuffer? In other words, what happens if a texture is used as both a source for texturing and a destination for rendering?*

RESOLUTION: resolved, (b2) - results are undefined because the framebuffer is not "framebuffer complete".

Originally this was resolved as causing framebuffer to fail the completeness test--i.e., rendering would be disabled (b1)

As background, the reason this is an issue in the first place is that simultaneously reading from, and writing to, the same texture image is likely to be problematic on multiple vendors' hardware without paying performance penalties for excessive synchronization and/or data copying.

There are, however, certain cases where this functionality would arguably be useful, supportable, and well-defined. In particular, we can consider the case of custom mipmap generation using one level's image as source data to render into other levels of the same texture.

So, at a minimum, we would like to support rendering to a currently bound texture object if the source texture object has the BASE_LEVEL and MAX_LEVEL texture parameters set such that the level being used as a framebuffer-attachable image is excluded from texture fetches.

This was our original rationale:

a1) is problematic because one context could modify the base/max level on a shared texture causing another context which is using the texture as a destination to throw an error. This idea was rejected as it essentially meant that the error would need to be thrown at render timer which people found unacceptable.

b1) has the same kind of multicontext behavior but no error. One context can cause a framebuffer shared in another context to become invalid, but this is already true and can happen for a variety of reasons if the participating framebuffer-attachable images and/or framebuffer attachments are modified by either context.

At the time, we also considered the following questions: should the specification require the framebuffer to fail the framebuffer completeness test? Or is the framebuffer simply "allowed" to not be complete in this case? The latter choice would imply that the framebuffer might still be considered "framebuffer complete" on some implementations. See issue (46)

c1) is the easiest to specify and has an advantage that some implementations may be relying on this behavior already. However, this was rejected as it is the least portable of the three options.

We originally chose option (b1), though we considered that later on, individual hardware vendors may offer layered extensions that change this "framebuffer completeness" failure into a success with either defined or undefined rendering behavior.

However, this issue was re-opened becaues the subsequent resolution of issue (66) was that there should be no "context-dependent" reasons for framebuffer incompleteness. If we had stuck with option (b1), then we would be making the framebuffer completeness predicated on a piece of context state (the current texture binding). Consider the case where texture T is attached to a framebuffer. Then this would have meant that a framebuffer could be complete in one context (that didn't have texture T bound as a texture) and incomplete in another context (that did have texture T bound).

When reconsidering this issue, we realized that we would not throw an error at Begin time without disabling rendering, so we really only considered the following revised set of options:

a2) throw an error and disable rendering, but don't affect framebuffer completeness
b2) the behavior is undefined

The issue was resolved the second time as:
b2) Undefined behavior

Another option that was briefly considered was to make this another type of error (unrelated to trying to render with an incomplete framebuffer). However, part of the rationale for throwing an error at glBegin time when trying to render with an incomplete framebuffer was that if you already have to test for framebuffer completeness, then throwing an error is no additional implementation burden. Yet, since it was decided that the "texture-from-destination" condition is not part of framebuffer completeness - issue (66) - then it is an additional burden to perform the "texture-from-destination" check just so that an error can be generated. The concern was some implementations might not need to check for this case at all and we didn't want to burdern those implementations with an additional Begin-time error check.

Also, for what it's worth, if we had left the "texture-from-destination" case in the framebuffer completeness test then any language describing how framebuffer completeness is affected when a currently bound texture is used as both source and destination needs to be explicit that the texture has to be currently bound *and* enabled. For instance, consider the case where a user has a cubemap texture object name N bound to unit X and a 2D texture object name M also bound to unit X. What if the user would like to use the 2D texture M as a source while rendering to the faces of the cubemap texture N? We would like to support this scenario, so the language about a currently bound texture object would have needed to take the target into account. And to make matters more interesting, this means we would have needed to take texture enables and fragment shaders into account in this decision. In the end, we decided that "context-state" would not affect the defintion of framebuffer completeness we avoided this complexity (or at least moved it out of the framebuffer completeness test).

(45) Are framebuffer configurations with no color attachments allowed?

RESOLUTION: resolved, yes

The reason this is an issue is that the GL spec assumes there is always a color buffer. If a framebuffer with no images attached to any of the color buffer attachment points can be "framebuffer complete", then the core GL spec will need to be modified to relax the assumption that a color buffer always exists.

However, since one of the possible likely uses of this extnesion is to support depth texture rendering and stencil rendering for shadowing techniques, it seems like requiring an unused "dummy" color buffer in some cases is both inconvenient and a waste of memory.

Therefore, framebuffers do not require color attachments to be valid. Perhaps though we should require that a framebuffer with *no* attachments is invalid.

It also should be stated that attempting to render without the "appropriate" buffers attached needs to be defined. For instance, presumably, for depth rendering with no depth buffer attached, the depth test is disabled, as it is in traditional GL.

*(46) In the framebuffer completeness criteria, this extension introduces the idea that rendering can fail for implementation dependent reasons. Framebuffer completeness also considers implementation *independent* reasons for failure.*

Do we need to make special distinction between the cases where a framebuffer is not complete because of implementation dependent or because of implementation indepenent reasons?

RESOLUTION: resolved, yes, though this is really tied into how we resolve the minimum requirements for supporting this extension. See issue (61)

Examples where a framebuffer may be incomplete on some implementations but not others include:

- 16 bit z-buffer used with 8 bit stencil buffer
- 32 bit color buffer with 16 bit depth buffer
- others?

Examples where framebuffer MUST be incomplete on all implementations include:

- color-renderable image attached to a non-color attachment point

- depth-renderable image attached to a non-depth attachment point

- stencil-renderable image attached to a non-stencil attachment point

- all images attached to a framebuffer do not have the same dimensions

- multiple render targets of different bit depths

- texture image attached to the framebuffer is part of a currently bound and enabled texture and the image is within the range of mipmap levels that can be fetched by rendering.

To make this determination we need to describe the criteria we should use to determine whether a framebuffer *can* or *must* be incomplete.

The arguments for putting state vectors into the "can" fail case is that a later extension can come along and simply relax those portions of the framebuffer completeness definiton with no additional API. State vectors classified as "must" fail cases would at least require the later extension to add an additional enable to start passing.

(47) Certain state-modification operations can cause a change to the validated state of a framebufffer. (I.e., can make a framebuffer that was complete become incomplete, or vice-versa). Do we want to list exactly which state-modification routines can cause this to happen? If so what is the list?

RESOLUTION: resolved, the answer is: yes we want to delineate exactly which routines can cause validation state changes.

Currently any routine which changes any of the following state can potentially cause framebuffer completness to change:

```
framebuffer state
state changes to attached objects
currently bound fragment program
texture enable state
```

The list of operations that can cause framebuffer a change to framebuffer completeness are spelled out in section 4.4.4.2.

(48) What information should be returned from CheckFramebufferStatusEXT()?

New RESOLUTION: resolved: 8 possible enum values, see issue (55)

```
Previous RESOLUTION: resolved, return one of three
enumerated values:
    1. GL_FRAMEBUFFER_COMPLETE_EXT
    2. GL_FRAMEBUFFER_UNSUPPORTED_EXT
    3. GL_FRAMEBUFFER_INCOMPLETE_EXT
where the three values mean the following:
    1. framebuffer is complete and supported
    2. framebuffer is not supported for implementation
       *dependent* reason
    3. framebuffer is incomplete for    implementation
       *independent* reason

We considered the following two sets of enums:
    Set 1:
        GL_FRAMEBUFFER_COMPLETE_EXT
        GL_FRAMEBUFFER_NOT_COMPLETE_EXT
        GL_FRAMEBUFFER_NOT_SUPPORTED_EXT

    Set 2:
        GL_FRAMEBUFFER_COMPLETE_EXT
        GL_FRAMEBUFFER_INCOMPLETE_EXT
        GL_FRAMEBUFFER_UNSUPPORTED_EXT

New resolution is Set 2.

NOTE: In order to fully resolve issue (55), we expanded this set
of enums to identify all of the implementation-independent
causes for a failure of the framebuffer completeness test.

Originally, we had decided to have a query where the
query returns one of three possible values

One possible set of names that could be returned included:
    FRAMEBUFFER_COMPLETE, and
    FRAMEBUFFER_HW_DEPENDENT, and
    FRAMEBUFFER_HW_INDEPENDENT

How much information we return from CheckFramebufferStatus
is a function of how we expect the return value to be used.
A framebuffer object that is not complete for implementation
*indepednent* reasons is really an indication of a
programming error (like mismatched sizes) and should only
occur during development phase of an application.  The
correct response to this failure is to modify the
application to fix the bug.  After application development,
a framebuffer object that is not complete for implementation
*dependent* reasons is possible.  However, it's not yet
clear whether we can easily characterize these reasons for
failure in a programmatic fashion that would really offer
the application enough information to do something different
at runtime.  Perhaps a human readable info log, intended
just as an application debugging aid, would be more
appropriate.

We also considered whether we needed two separate queries:
One that queried whether the framebuffer was complete
according to the spec, and one that queried whether the
```

framebuffer was supported. This was a little problematic as it might not be possible to answer the "IsFramebufferSupported" query until the framebuffer was complete. A possible solution would have been to return UNKNOWN from the "IsFramebufferSupported" query until the "IsFramebufferComplete" query returned TRUE.

In any event, we decided a single query was a simpler solution.

In addition, the proposed "format group / format restriction" API (see issue 12) should make the implementation-dependent framebuffer incomplete case much less likely (and perhaps impossible) to occur.

Note that if a framebuffer's state violates more than one of the framebuffer completeness rules described in section 4.4.4.2, then it is undefined which of the enumerated value corresponding to one of the violated rules will be returned by CheckFramebufferStatusEXT. Since the initial state of a framebuffer violates multiple rules from section 4.4.4.2, it is therefore undefined exactly which value is returned if CheckFramebufferStatusEXT is called while bound to a newly created framebuffer object.

(49) *When this extension is used in conjunction with MRT (multiple render targets), it would naively be possible to create a framebuffer that had different color bit depths/formats for various color attachment points. Should this be allowed?*

RESOLUTION: resolved, no, not in this extension.
A soon to follow extension may add this feature.

This feature could be supported by simply not requiring that all of the FRAMEBUFFER_COLOR_ATTACHMENTn images share the same internal format. We decided against doing so, however.

ARB_draw_buffers and OpenGL-2.0 do not provide any mechanism to support rendering to multiple color buffers of different formats. Consequently, we chose not to extend OpenGL in this manner as part of the EXT_framebuffer_object extension.

Presumably, a future layered extension could easily add this feature. There are some open questions about exactly how this might work. For instance, what should a query of RED_BITS return if the attached color-renderable images have different formats? In any event, we leave the details of rendering to differently formatted MRT for a future extension to define.

(50) *This extension introduces the concept of attaching one GL object (texture, renderbuffer) to another GL object (framebuffer). In many ways this situation is analogous to a previously poorly specified situation where a GL object could be attached to multiple contexts and the issues this raises with deletion and state propogation are similar. Several issues resolutions have been predicated on the assumption that*

as we specify this container/member relationship, the generation of GL errors should never be triggered in one context based on the asychronous actions of another context. Is this a valid premise?

In other words, should we be using the prevention of asynchronously generated GL errors as a design constraint?

RESOLUTION: resolved, no.

We didn't officially decide on this as a design constraint. However, we essentially decided it by proxy. We decided in issue (26) and (66) that an incomplete framebuffer can cause GL errors on rendering or reading the framebuffer. Consequently, this means a framebuffer shared by two contexts can be made incomplete by either context, and therefore each context can effectively cause the other context to start generating errors asynchronously.

We would expect that the state of framebuffer completness, like all the state of all shared objects, is not "guaranteed" to show up in another context until that context makes an "atomic" request to the server (like a BindFramebuffer for instance). Until that point, it is undefined whether the state change will show up in the other context, just like any state change made on a shared texture object.

(51) What api should we use to query the attachments of the currently bound framebuffer?

RESOLUTION: resolved, (b)

This is an issue because the relevant state for a specific attachment point is a function of the type of object attached to that that attachment point. The attachment point state needs to select a an image from an object which may have a collection of images, for instance the faces of a cube map texture.

This introduces a kind of "polymorphism" into the framebuffer attachment point that is problematic for queries.

We have a few options:

a) Some kind of single atomic query that returns a variable number of values in an array:

```
   GetFramebufferParameteriv(enum target,
                             enum pname,
                             int* params);
        where
        <target> = a framebuffer target
        <pname> = {attachment_point}
```

```
                Upon success "params" will contain an array of
                values where

                params[0] = {NONE | TEXTURE | RENDERBUFFER}

                if params[0] == TEXTURE then
                    params[1] = texture object name
                    params[2] = level
                    params[3] = face
                    params[4] = image
                else if params[0] = RENDERBUFFER then
                    params[1] = renderbuffer name

                Elements of the params array not explicitly defined
                above will have undefined values.

           One problem with (a) is that we would potentially also
           need a query to identify how many state variables will
           come back in this query.  Consider the case where in the
           future we add a new attachable object type that needs
           more selections tate or even add new selection state to
           existing object types.  Applications coded to expect a
           maximum of n values returned today may break in the
           future unless they have a way to dynamically learn how
           many attachment state params will come back from the
           query.

        b) individual queries for all the possible attachment
           state values.

           We create a new routine to add a new <attachment>
           argument, otherwise we'd have an explosion of
           permutations of attachment points and possible attachment
           selection state values

           This could look like

           void GetFramebufferAttachmentParameteriv(enum target,
                                                    enum attachment,
                                                    enum pname,
                                                    int *param);

                where
                <target> = a framebuffer target
                <attachment> = {attachment_point}
                <pname> = one of
                    FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE_EXT
                    FRAMEBUFFER_ATTACHMENT_OBJECT_NAME_EXT
                    FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL_EXT
                    FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE_EXT
                    FRAMEBUFFER_ATTACHMENT_TEXTURE_3D_ZOFFSET_EXT

                Upon success, param will be filled out as follows:

                if pname is FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE_EXT,
                then param will contain one of:
                    { NONE | TEXTURE | RENDERBUFFER },
                else if pname is
                FRAMEBUFFER_ATTACHMENT_OBJECT_NAME_EXT, and
                FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE_EXT = TEXTURE,
                then param will contain:
                    { name of attached texture }

                else if pname is
                FRAMEBUFFER_ATTACHMENT_OBJECT_NAME_EXT, and
                FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE_EXT =
                RENDERBUFFER, then param will contain:
                    { renderbuffer object name }

                else if pname is
                FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL_EXT, and
                FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE_EXT = TEXTURE,
                then param will contain:
                    { selected mipmap level of attached texture }

                else if pname is
                FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE_EXT,
                and FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE_EXT =
                TEXTURE, then param will contain:
                    { selected face of attached cube map texture }
                    { 0 if texture target is not TEXTURE_CUBE_MAP }

                else if pname is
                FRAMEBUFFER_ATTACHMENT_TEXTURE_ZOFFSET_EXT, and
                FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE_EXT = TEXTURE,
                then param will contain:
                    { selected z-slice/image of attached 3D texture }
                    { 0 if texture is not 3-dimensional }

                otherwise, param will contain the value 0.
```

One problem with option (b) is that it is a little heavy-handed as every piece of state needs its own query and enum

Given the above choices, and the problems of extending option (a) in the future, (b) is probably the better of the two choices. It really only adds a few enums, and though it does require an independent function call to obtain each piece of state, this is well-precedented behavior throughout GL.

(52) Should manual mimpap generation via GenerateMipmap apply to textures regardless of whether they are attached to framebuffer objects? Should automatic mimpap generation apply to all textures regardless of whether they are attached to framebuffer objects?

RESOLUTION: resolved, (a) - both apply to both.

This is an issue because the introduction of GenerateMipmap is intended both to address long standing complaints about the existing "automatic" mipmap generation API and to provide a

```
        clear trigger for render to texture API's to know when to do
        the mipmap generation.

        These API's could be considered completely orthogonally.  It's
        clear how they could interoperate.  The question is should they
        interoperate, or should one supercede the other?

        There are a couple of ways to address this issue:

           a) "automatic" mipmap generation applies always and is
               triggered by any gl{Copy}Tex{Sub}Image call if
               GENERATE_MIPMAP is set to TRUE.  "Manual" mipmap
               generation applies always and is triggered by a call to
               GenerateMipmaps

           b) "automatic" mipmap generation applies only
              to textures which are not attached to framebuffer
              objects, calls to GenerateMipmap on "unattached" textures
              are ignored.

              "manual" mipmap generation applies only to textures which
              are attached to framebuffer objects, the value of
              GENERATE_MIPMAP for "attached" textures is ignored

          c) Like option (b), but allow GenerateMipmap to
             apply to all textures and only let automatic mipmap
             generation apply to "non-attached" textures.

          d) Create an enable or other piece of state
             to toggle between allowing automatic and allowing manual
             generation.

       We disregarded (d) because it's not clear why an application
       that had the freedom to set this new enable bit wouldn't
       simply just turn off the legacy automatic mimpap generation
       to start with.

       Of the remaining choices, (a) is the most "orthogonal".  The
       intent of adding GenerateMipmap is to provide a cleaner and
       saner interface to mipmap generation that we would encourage
       developers to use over the automatic method.  Given that, it
       seems like restricting the "manual" generation to certain
       cases doesn't serve that goal, so we wish to allow its use
       on any textures, attached or not.
```

(53) When supporting ARB_draw_buffers, do we need the level of indirection between fragment color outputs and attached mages provided in that API?

```
           RESOLUTION: yes

           ARB_draw_buffers allows the user to set up an "indirection
           table" between the fragment color outputs ("result.color[n]"
           in ARB_fragment_program, and "gl_FragData[n]" in GLSL) and
           the attached draw buffers (FRONT, BACK, LEFT, RIGHT, etc).

           Since EXT_framebuffer_object is creating new non-visible
```

```
            framebuffer objects for which the legacy attachment points
            may not be appropriate, we could consider naming the
            attachment points by numerical index (COLOR0 ... COLORn) in
            which case we could consider dropping this level of
            indirection and allowing the fragment shader to output
            directly into the numerically specified COLOR0 attachment
            point with no indirection.

            However, this indirection is deemed to be useful becaues it
            allows the application to redirect the fragment color
            outputs without changing either the fragment shader itself
            or the current framebuffer attachments, both of which are
            believed to be heavier-weight state change operations than
            simply changing the indirection table via glDrawBuffers..

            Therefore, we elect to retain this level of indirection.
            This leaves open the question of what to call the attachment
            points.  See issue (54).
```

(54) What should we name the logical buffer attachment points, bearing in mind the relationship to ARB_draw_buffers?

```
            RESOLUTION: resolved, option (E), which is a modified
            version of (C).  Specifically, we use the names
            COLOR_ATTACHMENT0_EXT through COLOR_ATTACHMENTn_EXT,
            DEPTH_ATTACHMENT_EXT, and STENCIL_ATTACHMENT_EXT (and any
            future attachment points also get the ATTACHMENT suffix).

            The reason this is an issue is that prior to
            EXT_framebuffer_object, the names of the various color
            logical buffer "attachment points" were heavily influenced
            by their intended usage in a graphical window-system.
            Logical buffers for BACK and FRONT_LEFT make sense in the
            context of double buffering and stereo presentation, but
            their use in off-screen rendering situations is
            anachronistic at best and perhaps even confusing.

            There are several options:

            Option (A): stick with the "legacy" names

                This would have us use all of the legacy names which are
                used to identify a single buffer: FRONT_LEFT,
                FRONT_RIGHT, BACK_LEFT, BACK_RIGHT, and AUX0..AUXn.

                This option would require no change to DrawBuffersARB().

            Option (B): AUXn names

                If we wish to avoid using the legacy names, one option
                is to re-use another numerically named set of color
                buffers, the AUX buffers, and only allow framebuffer
                objects to support AUX0..AUXn attachment points.

                This has the advantage of being easy to specify, and
                numerically delimit, but is a little strange as
                framebuffer objects could conceivably support the same
```

```
        number of AUX buffers as the implementation supports
        multiple render targets.  This would have the awkward
        consequence of allowing framebuffer objects to support
        more AUX buffers than the default framebuffer could
        support via the pixel format selection mechanism.

        This option would require no specific change to
        DrawBuffers but might require non-default framebuffers
        to support more AUX buffers than the default framebuffer
        controlled by the window-system pixel format.

    Option (C): COLORn names

        Another option is to rename the color buffer attachment
        points for application-created framebuffer objects to
        COLOR0..COLORn.  This has the advantage of avoiding the
        legacy window-centric names, and avoiding the confusion
        with AUX buffers.  When considered in conjunction with
        the decision in issue (53) to support a level of
        indirection when using ARB_draw_buffers, however, the
        user of the names COLOR0..COLORn may be confusing.  For
        instance, if an ARB fragment program contains color
        output to "result.color[3]", it will not necessarily
        output to COLOR3.  It will actually write to the buffer
        specified by DrawBuffersARB for DRAW_BUFFER3 which may
        or may not be COLOR3.

        This option would require an update to DrawBuffers to
        accept the new COLOR0..COLORn values as valid draw
        buffers and would require a change to DrawBuffers to
        disallow the "legacy" names.  Or at the very least we
        would need some language to describe what happens if the
        DrawBuffers are using the legacy names when the
        currently bound framebuffer is not the default window
        system framebuffer.

    Option (D): DATAn names

        Yet another option is to call these attachment points
        DATA0..DATAn.  This is the same as option (C) but uses
        the word DATA instead of COLOR.  This has the advantage
        of avoiding the above problems with COLOR0..COLORN, but
        introduces a similar conflict with GLSL which uses the
        "gl_FragData[n]" name for its output.  Additionally,
        since we only support multiple render targets for color
        logical buffers, it may be that using the word DATA is
        considered too abstract/general.

    Option (E): add ATTACHMENT to *ALL* names

        In order to avoid the confusion of option (C) and (D),
        we can choose to be more verbose.  We can add the word
        _ATTACHMENT to distinguish these enums from the color
        outputs of a fragment program or fragment shader.  For
        symmetry we also add _ATTACHMENT to the DEPTH and
        STENCIL (and any other to-be-added) attachment points.
```

Similar to (C) and (D), this option requires an update to DrawBuffers to at least accept the new enum values. We could choose to make it illegal to specify the legacy values for non-default framebuffers as well. This is essentially covered by issue (55).

Here is an pseudo-code example using option (E):

```
// Assume presence of color renderbuffers
// with names 1000, 2000, 3000, 4000

GLuint db[4] =
    { COLOR_ATTACHMENT4, COLOR_ATTACHMENT7,
      COLOR_ATTACHMENT1, COLOR_ATTACHMENT2 };

glDrawBuffers(4, db);

glFramebufferRenderbufferEXT(GL_FRAMEBUFFER_EXT,   COLOR_ATTACHMENT4,
                             GL_RENDERBUFFER_EXT, 1000);

glFramebufferRenderbufferEXT(GL_FRAMEBUFFER_EXT,   COLOR_ATTACHMENT7,
                             GL_RENDERBUFFER_EXT, 2000);

glFramebufferRenderbufferEXT(GL_FRAMEBUFFER_EXT,   COLOR_ATTACHMENT1,
                             GL_RENDERBUFFER_EXT, 3000);

glFramebufferRenderbufferEXT(GL_FRAMEBUFFER_EXT,   COLOR_ATTACHMENT2,
                             GL_RENDERBUFFER_EXT, 4000);

Then in ARB_fragment_program
    result.color[0] writes to COLOR_ATTACHMENT4  (i.e., renderbuffer 1000)
    result.color[1] writes to COLOR_ATTACHMENT7  (i.e., renderbuffer 2000)
    result.color[2] writes to COLOR_ATTACHMENT1  (i.e., renderbuffer 3000)
    result.color[3] writes to COLOR_ATTACHMENT2  (i.e., renderbuffer 4000)

And in ARB_fragment_shader
    gl_FragData[0]  writes to COLOR_ATTACHMENT4  (i.e., renderbuffer 1000)
    gl_FragData[1]  writes to COLOR_ATTACHMENT7  (i.e., renderbuffer 2000)
    gl_FragData[2]  writes to COLOR_ATTACHMENT1  (i.e., renderbuffer 3000)
    gl_FragData[3]  writes to COLOR_ATTACHMENT2  (i.e., renderbuffer 4000)
```

See also issue (57) for discussion on querying the number of available color buffers.

(55) What should happen if the current DRAW_BUFFER(s) point to a non-existent logical buffer? Likewise for READ_BUFFER.

RESOLUTION: resolved

partial resolution #1: DrawBuffer(s)/ReadBuffer throws an error if the buffer does not "exist" for all framebuffers (default and non-default).

Should it be an error to call drawBuffer on a non-default framebuffer if named buffer does not exist?

Resolved: yes

partial resolution #2: The test for having a valid draw and read buffer should be part of framebuffer completeness test.

```
            Should be part of completeness test and should all 5
            indpenent reasons add 5 enums?

                Resolved: yes

        partial resolution #3: we should create an enum for
        each implementation independent reason for failing
        the framebuffer completeness test of section 4.4.4.1

            Current names (could change...)
                FRAMEBUFFER_COMPLETE_EXT
                FRAMEBUFFER_INCOMPLETE_ATTACHMENTS_EXT
                FRAMEBUFFER_INCOMPLETE_IMAGES_EXT
                FRAMEBUFFER_INCOMPLETE_DIMENSIONS_EXT
                FRAMEBUFFER_INCOMPLETE_FORMATS_EXT
                FRAMEBUFFER_INCOMPLETE_DRAW_BUFFER_EXT
                FRAMEBUFFER_INCOMPLETE_READ_BUFFER_EXT
                FRAMEBUFFER_UNSUPPORTED_EXT

                NOTE: as per resolution of issue (78)
                FRAMEBUFFER_INCOMPLETE_ATTACHMENTS_EXT
                became
                FRAMEBUFFER_INCOMPLETE_ATTACHMENT_EXT
                and
                FRAMEBUFFER_INCOMPLETE_IMAGES_EXT
                was dropped.
```

This issue is intertwined with issue (56), which discusses whether the DRAW_BUFFER and READ_BUFFER are context or framebuffer object state.

First, some background: If DRAW_BUFFER state is part of the context state vector rather than the framebuffer object state vector (see issue 56), then there are three ways to cause DRAW_BUFFER to reference a color buffer attachment point that "does not exist" in the currently bound framebuffer. If DRAW_BUFFER is part of the framebuffer object state vector, then (A) still applies but (B) and (C) do not.

A) The first case is by detaching, from the currently bound framebuffer object, the image that is attached to attachment point named by the value of DRAW_BUFFER. If an image is attached to COLOR_ATTACHMENTn_EXT in the current framebuffer object and DRAW_BUFFER is set to COLOR_ATTACHMENTn_EXT, and then the application detaches the image from COLOR_ATTACHMENTn_EXT, then DRAW_BUFFER will end up specifying a buffer that "does not exist" in the currently bound framebuffer object.

   There is no analogue to this case in OpenGL prior to EXT_framebuffer_object. Before this extension, the pixel format or fbconfig of a window or pbuffer is immutable once one of these drawables has been created. By design, framebuffer objects (which essentially represent a new type of drawable) have mutable "pixel formats".

B) The second case is by binding between two user-created framebuffer objects, where the two framebuffer objects do not have images attached to the same set of color attachment points. If an image is attached to COLOR_ATTACHMENTn_EXT in the current framebuffer object and DRAW_BUFFER is set to COLOR_ATTACHMENTn_EXT, and then the user binds to a new framebuffer for which there is no image attached to COLOR_ATTACHMENTn_EXT, then DRAW_BUFFER will end up specifying a buffer that "does not exist" in the newly bound framebuffer object.

This is morally equivalent to calling MakeCurrent to bind a context to a different drawable (window or pbuffer) which does not have bitplanes for the color buffer named by the context's value of DRAW_BUFFER. For example, MakeCurrent to a double-buffered window, set DRAW_BUFFER to BACK, then MakeCurrent to a single-buffered window.

C) The third case is by binding between the default framebuffer and a user-created framebuffer object. The attachment points of a user-created framebuffer object are named COLOR_ATTACHMENTn_EXT, DEPTH_ATTACHMENT_EXT, STENCIL_ATTACHMENT_EXT, etc. These are also the legal values of DRAW_BUFFER when a user-created framebuffer object is bound. The default framebuffer, on the other hand, does not use the _ATTACHMENT names but instead uses names such as FRONT_LEFT, BACK_RIGHT, and AUXn as legal DRAW_BUFFER values. Because the two sets of names do not overlap, no value of DRAW_BUFFER is valid for both the default framebuffer and a user-created framebuffer object.

This is somewhat equivalent to case (B), except that in case (C) there is a guarantee that DRAW_BUFFER will become invalid, whereas in case (B) it is only _possible_ that DRAW_BUFFER will become invalid.

The very problem of invalid DRAW and READ buffers was already a feature of OpenGL (and the window-system APIs) before the introduction of the EXT_framebuffer_object extension. The GLX specification specifically addresses what happens when MakeCurrent is used to bind a context to a different drawable (window or pbuffer) which does not possess one of the color buffers referenced by the context's current values of DRAW_BUFFER and READ_BUFFER. GLX addresses this by saying that no GL error is generated, but invalid DRAW_BUFFER behaves as if DRAW_BUFFER were NONE, and reads produce undefined results when READ_BUFFER is invalid.

Now, back to the question of how EXT_framebuffer_object should handle the situation when a framebuffer object is bound and DRAW_BUFFER or READ_BUFFER is not valid while bound to a user-created framebuffer object.

Obviously one option is to resolve the issue the same way is handled by MakeCurrent in the GLX spec. Invalid DRAW_BUFFER acts as if DRAW_BUFFER were NONE, and invalid READ_BUFFER causes read operations to generate undefined results.

A second option is to modify the framebuffer completeness test to fail if the current DRAW_BUFFER or READ_BUFFER reference an attachment point to which no image is attached. This solution would also result in no rendering being performed, but would also generate a GL error when rendering is attempted while in this state, as determined by issue (64). When rendering to a framebuffer object, invalid DRAW_BUFFER would cause generation of GL errors; but when rendering to a window, invalid DRAW_BUFFER would not cause generation of GL errors.

Consider also that, because of the resolution of issue (66), depending on how issue (56) is decided, failing the framebuffer completeness test due to a "non-existent" DRAW_BUFFER or READ_BUFFER may not be a viable option, because the framebuffer completeness test is not allowed to examine context state.

Additionally, there are two sub-issues that fall out of this issue:

    sub-issue 1: Error at DrawBuffer call time or not?
    sub-issue 2: DRAW_BUFFER in or out of completeness test?

[sub-issue 1]: First, what should be the behavior of DrawBuffer(s) and ReadBuffer if the specified buffer does not exist at the time DrawBuffer(s) or ReadBuffer is called?

For default framebuffer (window-system drawables), an error is currently thrown. We can not (or do not wish to) change this legacy behavior of window-system supplied drawables. Consequently, we must resolve several questions here:

For instance:

- Should we do the same thing (error at DrawBuffer time) for user framebuffer objects?

- Is this decision influenced by the fact that user-created framebuffer objects can change their attachments one buffer at a time while window-system supplied drawables can not (i.e., must change all attachments atomically)?

- Also, on other places in this API, such as assembling a framebuffer from framebuffer-attachable images, we have allowed the system to move through "invalid" states without generating an error as long as the system was back in a "valid" state by rendering time (or "validation" time). Should we adhere to that principle here, or is this case different somehow?

- Do we wish to retain the legacy window-system DrawBuffer(s) behavior for application-created framebuffer objects for the sake of maintaining consistency? i.e., Does the benefit of treating default and non-default framebuffers consistently outweigh the earlier decision to delay validation of "invalid" states?

- Both resolutions are examples of "state combination" errors where an error may or may not be generated depending on the order state-changing function calls are made. For instance, in the legacy behavior DrawBuffers does or does not throw an error on user framebuffer objects depending on when you call DrawBuffer relative to when you made your image attachments. On the other hand, if we decided to not throw an error at DrawBuffer time for user framebuffer objects, then DrawBuffer does or does not throw an error depending on whether one is bound to a default or non-default framebuffer. Is one of these "state combination" errors better or worse than the other?

[sub-issue 2]: Should having a DRAW_BUFFER that names a non-existent buffer cause the framebuffer completeness test to fail?

Since image attachments can be changed after DrawBuffer(s) is called, even if we throw an error at Drawbuffer(s) time, we still must decide how to handle having an invalid DRAW_BUFFER at render (or "validation") time. Our options include failing the completeness test, (thus disabling rendering and generating an error at render time) or just behaving as if DRAW_BUFFER is NONE (thus disabling rendering but generating no error at render time).

If the answer is "fail completeness test", then since currently framebuffer completeness can only be affected by framebuffer state, then one of two things has to happen: Either the drawbuffer state must be framebuffer object state, or we have to revisit our decision that framebuffer completeness is solely a property of the framebuffer state and can not be affected by "per context" state.

If the answer is "do not fail completeness test", then the practical consequence of this decision is that having an invalid DRAW_BUFFER behaves as if DRAW_BUFFER is NONE, and no error is generated at render time. Also, in this case, DRAW-BUFFER state can be either per-context or per-framebuffer object state without violating any previously decided issues.

(56) Should the value of DRAW_BUFFER, the corresponding draw buffers indirection table for ARB_draw_buffers, and the value of READ_BUFFER, be part of the context state vector or part of the the framebuffer object state vector?

RESOLUTION: resolved, per-framebuffer object

This issue is intertwined with issue (55), which discusses what happens when the DRAW_BUFFER or READ_BUFFER references a color buffer that "does not exist" in the current framebuffer.

Please first read the "First, some background" section of issue (55), which could be, but is not, replicated here. Note that depending on how issue (56) is decided, cases (B) and (C) from issue (55) might become moot. Specifically, if the DRAW_BUFFER and READ_BUFFER state are added to the framebuffer object state vector, then neither case (B) nor case (C) remains relevant. Only case (A) would continue to be an issue.

The discussion over this issue centered around the following areas:

i) There must be a unique per-context value of DRAW_BUFFER for the default window-system-provided framebuffer.

In GL, before EXT_framebuffer_object, the DRAW_BUFFER was considered context state because:

1) When two contexts are rendering to the same drawable, each context can use a different value of DRAW_BUFFER.

2) When MakeCurrent alternately binds a single context to each of two different drawables, after MakeCurrent DRAW_BUFFER retains the value it had immediately before calling MakeCurrent. This is true even if the last time the context was bound to a given drawable, DRAW_BUFFER had a different value than it does when that drawable is next bound to the context.

Therefore, a per-context value of DRAW_BUFFER must exist, and must be in effect when the FRAMEBUFFER_BINDING_EXT is zero.

Two ways of satisfying this requirement that we have considered include:

A) DRAW_BUFFER is part of the context state vector, but is not part of the framebuffer object state vector.

B) Every framebuffer, including the per-context default window-system-provided framebuffer, has its own value for DRAW_BUFFER.

ii) MakeCurrent vs. BindFramebuffer

As described above, the context state vector must contain a value for DRAW_BUFFER that applies to the default window-system-provided framebuffer, which is used after a call to BindFramebuffer(0). When MakeCurrent is used to bind the context to a different drawable (window or pbuffer), the context's value of DRAW_BUFFER remains unchanged. In other words, the choice of drawable does not affect the value of DRAW_BUFFER.

An application-created framebuffer object is another type of drawable. When the framebuffer binding is changed via BindFramebuffer, issue (56) speaks to the way in which DRAW_BUFFER is or is not updated. If DRAW_BUFFER is part of the context state vector, then DRAW_BUFFER remains unchanged after calling BindFramebuffer, just like it remains unchanged after calling MakeCurrent. On the other hand, if DRAW_BUFFER is part of the framebuffer object state vector, then after calling BindFramebuffer DRAW_BUFFER may change along with the rest of the per-framebuffer state (i.e., the image attachments).

By defining DRAW_BUFFER as context state, the behavior of BindFramebuffer and MakeCurrent are similar, with respect to their effect on the value of DRAW_BUFFER.

On the other hand, by defining DRAW_BUFFER as framebuffer object state, then BindFramebuffer and MakeCurrent differ in their impact on the value of DRAW_BUFFER.

iii) Multiple contexts and shared framebuffer objects

If DRAW_BUFFER is part of the framebuffer object state vector, then a single value of DRAW_BUFFER, like all of the framebuffer object state, will be shared by any context bound to a given framebuffer object. This can be considered either a feature or a restriction depending on whether or not it is desirable for multiple contexts to be able to share a single the value of DRAW_BUFFER.

Note that WGL_ARB_pbuffer plus WGL_ARB_render_texture API has limitations due to the fact that the texture image selection state is stored in the pbuffer drawable. For example, that API does not support six different contexts (in six different threads) simultaneously rendering to the six faces of a cube map pbuffer. It offers no way to share the images without also sharing the pbuffer, and the pbuffer contains a single set of texture image selection state.

EXT_framebuffer_object differs from ARB_render_texture, however, however, in that EXT_framebuffer_object allows the same images of a texture to be attached to multiple framebuffer objects. Consequently, the above cubemap example can be implemented in EXT_framebuffer_object in one or two ways, depending on the resolution of issue (56):

1) Create six framebuffer objects. Attach a different face of a cubemap texture to each of the six framebuffer objects. Each of the six contexts binds to a unique framebuffer object. Technically, this option is available whether DRAW_BUFFER is context or framebuffer state. However, if it is context state, then there is no reason to create six framebuffer objects since the value of the DRAW_BUFFER will already be unique per context.

2) On the other hand, if DRAW_BUFFER is defined as context state, then a second option is available. Using a single framebuffer object, attach each face of the cube map texture to a different attachment point in the framebuffer object. Each of the six contexts binds to the same framebuffer object, but each context uses a different value of DRAW_BUFFER.

iv) Frequency of DrawBuffer calls:

Whether DRAW_BUFFER is part of context or framebuffer state will have an effect on how often one must call DrawBuffer after modifying framebuffer state.

If DRAW_BUFFER is part of the context state vector, then DRAW_BUFFER is guaranteed to become invalid after calling BindFramebuffer to switch between the default framebuffer and a user-created framebuffer object [i.e., this is case (C) in issue (55)]. DRAW_BUFFER may become invalid after switching between two user-created framebuffer objects if the framebuffer objects do not have images attached to the same set of color attachment points. When DRAW_BUFFER is invalid, it is necessary to call DrawBuffer to set DRAW_BUFFER to a valid value or else rendering is disabled.

If, on the other hand, DRAW_BUFFER is part of the framebuffer object state vector, then it should never be necessary to call DrawBuffer after calling BindFramebuffer. DRAW_BUFFER would only become invalid if an image was detached from the framebuffer, or if MakeCurrent bound the default framebuffer to a drawable with a different set of color buffers. (The latter was possible prior to this extension.)

Note that there are several state-modifying routines that may also need to get called after a framebuffer state change, like Viewport, Scissor, etc. We are not proposing that these other routines be part of framebuffer state. One could think of DrawBuffer as being similar to these other routines which you may also need to call when you bind between framebuffer objects.

On the other hand, some have questioned whether an invalid DRAW_BUFFER is really in the same class of problems as an out-of-bounds viewport or scissor because: 1) an invalid viewport or scissor never generates a GL error, and 2) prior to the EXT_framebuffer_object extension an invalid DRAW_BUFFER would generate INVALID_ENUM inside DrawBuffer.

v) Effect on framebuffer completeness test:

By resolution of issue (66), if the draw buffer is context state, then the fact that the draw buffer names a non-existent buffer can not affect the result of the framebuffer completeness test. Note that this still could be considered a "do not render" case, but would separate from the framebuffer completeness test.

If the draw buffer(s) and read buffer are part of the framebuffer object state then having a draw or read buffer name a non-existent buffer can (if we choose) be part of the framebuffer (in)completeness test.

Note, by resolution of issue (64), failing the framebuffer completeness test causes a GL error to be generated when draw or read operations are attempted. Prior to EXT_framebuffer_object, it was already possible to have an invalid value of DRAW_BUFFER if a call to MakeCurrent bound the context to a drawable that did not contain a color buffer corresponding to the context's value of DRAW_BUFFER. However, no GL error would be generated if DRAW_BUFFER obtained an invalid value through this method.

vi) Draw buffer(s) error behavior:

Prior to the EXT_framebuffer_object extension, it was an error to call DrawBuffer or ReadBuffer with a value that did not correspond to one of the logical color buffers of the currently bound drawable (window or pbuffer). Although it was not possible to set DRAW_BUFFER to an invalid value by calling DrawBuffer, it was actually possible for DRAW_BUFFER to have an invalid value after a call to MakeCurrent, as describe in issue (55).

It has not been decided yet whether EXT_framebuffer_object will relax the requirement that the argument to DrawBuffer references a color buffer that "exists" in the currently drawable.

In working group discussions, there was a perception that such an error during DrawBuffer can be generated only if DRAW_BUFFER is part of the framebuffer object state vector. Then when the default framebuffer (window or pbuffer) is current, the legal values of the argument to DrawBuffer would be determined by the pixel format or fbconfig. When a user-created framebuffer object is current, the legal values of DrawBuffer would either be any of the COLOR_ATTACHMENTn_EXT names or only the names of attachment points to which an image is presently attached.

However, given the precedent set by MakeCurrent and DRAW_BUFFER, it seems reasonable to retain the preexisting requirement that the argument to DrawBuffer names a buffer that "exists" in the current drawable. In other words, there already exists precedent that says it is OK for DrawBuffer to generate an error in all the cases described in the preceeding paragraph, even if DRAW_BUFFER is defined as part of the context state vector.

(57) Should we have a query to define the maximum number of attachable color buffers (to support ARB_draw_buffers)?

RESOLUTION: yes, MAX_COLOR_ATTACHMENTS.

Currently an application can query the GL for the maximum number of supported AUX buffers. An application can also query for MAX_DRAW_BUFFERS_ARB in the ARB_draw_buffers extension. Given that we have named the color logical buffer attachment points, COLOR_ATTACHMENT0_EXT through COLOR_ATTACHMENTn_EXT, it seems natural that we should have a query to find the maximum value "n".

One thought was that we might be able to use MAX_DRAW_BUFFERS_ARB to store this value, but that value really describes the maximum number of colors that can be simultaneously output which is not the same thing as the number of buffers which can be attached and then selected among using DrawBuffersARB().

This question is related to issue (54), which covers the names of the user-created framebuffer object color attachment points. Using the names COLOR_ATTACHMENT0_EXT through COLOR_ATTACHMENTn_EXT rather than the legacy color buffer attachment names (FRONT_LEFT et. al.) for user-created framebuffer objects has an advantage that the number of color buffer attachment points could be queried independent of the number of AUX buffers and existence of front/back & left/right color buffers as specified in the pixelformat. The number of available offscreen attachment points really should be independent of the properties of the current drawable's pixelformat, especially since MakeCurrent can bind a context to a drawable with a different pixelformat and thus different set of color buffers.

One implication of this query is that the value of MAX_COLOR_ATTACHMENTS_EXT is possibly still dependent on the context/pixel format but independent of the currently bound framebuffer. In other words, MAX_COLOR_ATTACHMENTS_EXT can not change simply because the user called BindFramebuffer(). Or can it? See issue (62)

*(58) What should we do about rendering to textures with borders?
(besides attempt to fervently wish them out of existence, I mean)*

RESOLUTION: resolved, borders are fully supported

Should we allow rendering to textures with borders at all?
Resolved: yes

If we allow this, can you render to the border pixels?
Resolved: yes

The reason this is an issue is that (a) everyone hates supporting borders, and (b) it's not clear what it means to render to a texture with borders.

To disallow rendering to a texture image with non-zero border size, we could add a test for non-zero border size to the definition framebuffer completeness. This might be preferrable to an error at FramebufferTexture, since the user could always redefine the texture to have borders after attachment, and so the framebuffer completeness test is necessary anyway.

However, since borders do exist today and we are not planning to rip them out of OpenGL everywhere else, we decided to support them. It seemed odd that you could still specify borders via TexImage but not render into the same texture so we leave them supported. Note that it's quite possible that implementations which don't support borders may continue to either not support them or fall to software rasterization.

If someday we decide to disallow borders in general, they will be disallowed from this extension as well.

One additional note: section 3.8.2, page 137, of the OpenGL 1.5 specification, states that {Copy}TexSubImage uses negative offsets to refer to border texels. We choose not to do this because negative window-coordinates are undefined. (NOTE: Are negative window coordinates actually undefined? Or are they just not commonly used in practice?)

(59) Should we support named bit depths for stencil renderbuffers?

RESOLUTION: resolved, yes, choose 4 common formats.

We intend to support using renderbuffers to store stencil data. This means we need to consider what kind of "internal format" request we provide for stencil formatted renderbuffers.

We choose to allow a "named" format request for the internal format. This is essentially equivalent to the named internal format request of the TexImage calls. It is merely a request and the driver will attempt to satisfy it as best as possible but may approximate the requested format with another format. Additionally, this request is subject to the same invariance constraints as the texture internal format requests.

For the initial extension we choose the following four sized internal formats, as well as the base internal format STENCIL_INDEX:

```
    STENCIL_INDEX1_EXT
    STENCIL_INDEX4_EXT
    STENCIL_INDEX8_EXT
    STENCIL_INDEX16_EXT
```

(60) If depth buffer is disabled when a user-created framebuffer object is bound and an image is attached to GL_DEPTH, does the depth buffer factor into framebuffer validity determination or is the depth buffer ignored? Similar for other types of logical buffers.

RESOLUTION: resolved, consider all attached images when determining framebuffer completeness, even if the images are "irrelevant" based on the state of the framebuffer.

The main reason to consider not paying attention to certain images (i.e., ignoring the image attached to the depth buffer when depth test is disabled) would be developer convenience. The developer wouldn't need to explicitly detach a buffer, but could set the state to ignore it (disable depth test, or disable color mask, reset draw buffer, etc).

However, this raises the possibility that by simply changing this other state (depth test, stencil test, color mask, etc) the query for framebuffer completeness could change values. This was deemed undesirable. We'd like to be able to minimize the amount of state changes that can cause the framebuffer completeness query to change.

Another strange effect of ignoring "irrelevant" images when considering framebuffer completeness is that we could get an undesirable interaction between draw buffer and the pixel format for the framebuffer. A framebuffer is considered incomplete if the color buffers do not all have the same internal format. But, consider the following case:

- an application attaches a floating point color-renderable image to COLOR_ATTACHMENT1, and

- the application attaches a fixed point color-renderable image to COLOR_ATTACHMENT2 and

- the application sets the DRAW_BUFFER to COLOR_ATTACHMENT1, then

If we ignored the attached images not pointed to by DRAW_BUFFER(s} when evalutating framebuffer completeness, we could consider this framebuffer complete. This framebuffer would use floating point rendering. Now, if the application simply changes the DRAW_BUFFER to COLOR-ATTACHMENT2, then we would also say the framebuffer is complete but now the framebuffer would be using fixed point rendering. We didn't want to allow a change to DRAW_BUFFER to effectively change the pixel format. On the other hand if we always considered all attached images, then in this case described above, the framebuffer would always be incomplete while the formats of the color-renderable images were inconsistent.

To avoid the above complications, we choose to have framebuffer completeness queries consider all attached buffers, regardless of whether they would be "used" according to the current state vector or not.

(61) What are the "minimum requirements" to support this extension?

RESOLUTION: resolved, language added to end of 4.4.4.2

For instance, is it a requirement that there must be at least one renderable color, depth, and stencil format that can all work together? is it a requirement that you must be able to render to *any* "color-renderable" texture format?

Since this extension specifically pulling in functionality that used to be in the domain of the window sytem, we would like to use as a starting point for our requrirements, the language from the GLX 1.3 spec, page 15, which lists the minimum requirements langauge for a conformant GLX implementation.

Questions to answer:

- is the GLX spec a good starting point?

- do we want the same requirements as the GLX spec?

- do we want stronger requirements than the GLX spec?

- do we want some kind of requirement that states that to support this extension, there must be at least one "gl conformant" framebuffer configuration that can be constructed on a given implementation? If so, how do we phrase this?

Anyway, the GLX spec states:

"Servers are required to export at least one GLXFBConfig that supports RGBA rendering to windows and passes OpenGL conformance (i.e., the GLX RENDER TYPE attribute must have the GLX RGBA BIT set, the GLX DRAWABLE TYPE attribute must have the GLX WINDOW BIT set and the GLX CONFIG CAVEAT attribute must not be set to GLX NON CONFORMANT CONFIG). This GLXFBConfig must have at least one color buffer, a stencil buffer of at least 1 bit, a depth buffer of at least 12 bits, and an accumulation buffer; auxiliary buffers are optional, and the alpha

    buffer may have 0 bits. The color buffer size for this
    GLXFBConfig must be as large as that of the deepest
    TrueColor, DirectColor, PseudoColor, or StaticColor
    visual supported on framebuffer level zero (the main
    image planes), and this confguration must be available
    on framebuffer level zero."

So if we did a direct translation of these requirements into our spec, we'd end up with something approximately like the following:

    Although GL defines a wide variety of internal formats
    for textures and renderbuffers, some implementations may
    not support particular combinations of internal formats
    for the images attached to the framebuffer. For a
    framebuffer with these unsupported combinations of
    internal formats, calls to CheckFramebufferStatusEXT()
    will return FRAMEBUFFER_UNSUPPORTED_EXT.

    There must exist, however, at least one combinations of
    internal formats for the images attached to the
    framebuffer for which CheckFramebufferStatusEXT() will
    not return FRAMEBUFFER_UNSUPPORTED_EXT.

    Specifically, implementations are required to support at
    least one set of internal formats for the images
    attached to a framebuffer such that

        - the image attached to the color buffer supports
          RGBA rendering, and

        - the image attached to the color buffer has at
          least as many bits as the deepest visual supported
          by the window-system, although the alpha buffer
          can have 0 bits, and

        - the image attached to the depth buffer has at
          least 12 bits, and

        - the image attached to the stencil buffer has at
          least 1 bit, and

        - rendering to this framebuffer passes OpenGL
          conformance."

However, it looks like no one is seriously using the NON_CONFORMANT_CONFIG bit under GLX or AGL, and on WGL, there is no such bit, so we'd like to "assume" conformance and drop the last clause. Additionally, we'd like to just piggy back on the existing requirements without duplicating them here so we will simplify this language to leave out the last paragraph and list of clauses altogether.

We do wish to retain the notion that there must be some configuration for which FRAMEBUFFER_UNSUPPORTED_EXT is not returned.

(62) Exactly which, if any, queriable state can change after a call to BindFramebuffer and/or a change in framebuffer attachments?

RESOLUTION: resolved, at the Sept. 2004 ARB meeting we resolved in principle that there is a small subset of "framebuffer-related" state that can change. We just need to define exactly the subset. The current subset as listed in section 4.4.5 is below:

```
AUX_BUFFERS
MAX_DRAW_BUFFERS
MAX_COLOR_ATTACHMENTS
RGBA_MODE
INDEX_MODE
DOUBLEBUFFER
STEREO
SAMPLE_BUFFERS
SAMPLES
{RED|GREEN|BLUE|ALPHA}_BITS
DEPTH_BITS
STENCIL_BITS
ACCUM_{RED|GREEN|BLUE|ALPHA}_BITS
```

The reason this is an issue is that traditionally there are some GL context state queries that are dependent on pixel format and window-system state. For instance, doing a GetIntegerv of DEPTH_BITS returns the bit depth of the window-system allocated depth buffer which is a function of the pixel format. If DEPTH_BITS is zero, this means that no depth buffer was present in the pixel format. Other context state queries like MAX_DRAW_BUFFERS, MAX_ACCUM_BUFFERS, SAMPLES, etc are all possibly functions of the current pixel format, and have traditionally been constant over the lifetime of a given context.

However, this extension specifically subsumes some of the operations and state of the window-system pixel format mechanism. So an obvious question is: what should these queries return for things like DEPTH_BITS and MAX_DRAW_BUFFERS when using a non-default framebuffer object?

If we allow these queries to return a value that is a function of the current framebuffer object, then a consequence is that the values returned by these queries can change after a call to BindFramebuffer and/or a change in the attachments of the currently bound framebuffer object.

This may be desirable: for instance, a user may rightly expect that querying RED_BITS returns the red bits of the currently attached color buffer(s). But is the user also expecting that MAX_DRAW_BUFFERS might change? What about SAMPLES or SAMPLE_BUFFERS? What about MAX_COLOR_ATTACHMENTS?

Consider that in developing ARB_draw_buffers it was stated that some implementations might want to set MAX_DRAW_BUFFERS to 1 for pixel formats that also supported multisampling. This would allow implementations to control which capabilities they exported. What facilities do we have for this in this extension - can MAX_DRAW_BUFFERS change if we supported multisampling on a non-default framebuffer object?

Fundamentally, all of the state in table 6.28-6.31 of the OpenGL 1.5 spec (the MAX_* queries) can in theory change as the result of the pixel format changing. Since this extension does an effective pixel format change, what if any of this state can/should be allowed to change when framebuffer attachments are changed?

(63) Should we change ValidateFramebuffer into an explicit enum-based query for framebuffer completeness?

RESOLUTION: resolved, separate API function rather than a Get query, to emphasize the "on-demand" state examination.

We did choose a different name for ValidateFramebuffer(). In issue (67) we decided to rename this function CheckFramebufferStatus().

For reference the reason this is an issue is that, as originally described, ValidateFramebuffer (now called CheckFramebufferStatus) served three purposes:

First, it forced an "on-demand" examination of the current framebuffer state (including framebuffer attachment state) and the state of the attached images. On some implementations this examination might be expensive, and therefore there was a desire to control exactly when the operation would occur.

Second, because of the implementation dependent reasons that a framebuffer might be considered not complete, ValidateFramebuffer served as a query for an application to determine at run-time if a seemingly compatible combination of attached images is actually incompatible on the current GL implementation.

Third, ValidateFramebuffer was more than just a query. It was a function that would set a piece of framebuffer state that "enabled" rendering if the framebuffer was determined to be complete. After certain changes to framebuffer state, or in the initial default state, unless ValidateFramebuffer was called prior to rendering, and unless framebuffer validation "passed", rendering would be disabled.

However, now that it is no longer required to call ValidateFramebuffer prior to rendering, ValidateFramebuffer doesn't really set any state. The third reason is no longer pertinent.

This leaves us with the first and second reasons. The first reason in particular seems to be driven by convenience. It is convenient to be able to control when this operation happens, but it is arguably also convenient to be able to force the examination/validation of a wide variety of other pieces of GL state, yet we don't have specific on-demand "ValidateTexture" or "ValidateBlendState" routines. In addition, on some implementations framebuffer validation may be less expensive than originally thought.

So if we ignore the first reason for a moment, we are left the second reason for ValidateFramebuffer - a query of the framebuffer completeness. We do wish to retain this query somehow, so we could choose to leave it in its current form, or we could choose to make it look like other more traditional queries, i.e., some kind of GetInteger, GetFramebuffer, or GetFramebufferParameter call.

If we feel like the first reason is still valid, we could also choose to retain a ValidateFramebuffer call to get the "on-demand" state examination and still choose to make separate query for the framebuffer completeness.

Either way, if we decide to make an enum-based query we need to choose the form. We could choose to use GetInteger and query for COMPLETENESS. (If we do this, we'd need a "per-target" variant of the enum, i.e., FRAMEBUFFER_COMPLETE, and if a read framebuffer target is added later, READ_FRAMEBUFFER_COMPLETE would need to be added as well.) This would be similar to how texture bindings are queried on a per target basis as in GetIntegerv(TEXTURE_BINDING_2D, &param). Another option is to add a target-aware query routine, i.e., GetFramebufferiv(FRAMEBUFFER, COMPLETE, &param); this is similar to what the ARB vertex/fragment program API's did to query per-target state like PROGRAM_NATIVE_INSTRUCTIONS_ARB.

(64) Should it be a GL error to attempt to render with an incomplete framebuffer?

RESOLUTION: resolved, "YES"

In looking at other GL resources that can be considered "incomplete" for rendering, there were two precedents to draw on here: (a) textures and (b) programs/shaders.

a) For textures, the GL behaves as if the incomplete resource is simply not available. That is, if an application attempts to render with an incomplete texture, then the GL behaves as if texturing is simply disabled. No error is thrown.

b) For ARB_vertex_program and ARB_fragment_program, and GLSL shaders, if a program or shader is invalid, then the GL throws an error at "Begin" time.

Originally, we choose style (a): treat an incomplete framebuffer similar to a "pixel ownership test failure". This means that no fragments are generated, reads of the framebuffer generate undefined pixels, and no error is thrown.

[NOTE: Technically, according to the GL spec, the fate of rendered fragments that fail the pixel ownership test is left up to the window-system and is therefore implementation dependent. A better way to handle this is to mimic make_current_read's language "as if DRAW_BUFFER is NONE"]

However, since the a query of framebuffer completeness can only answer the question "is the framebuffer complete right now?", but doesn't indicate whether the application may have attempted to render with an incomplete framebuffer earlier, we decided to throw an error in this case as an aid to the developer. Throwing an error has an advantage in that the error state is retained, like all GL errors until the user calls GetError().

Another option that was considered was to extend the framebuffer completeness query to indicate that the framebuffer is complete now, but was incomplete during earlier rendering. The downside of this option was that then there would then be two return values for the query that would mean "framebuffer complete right now". So in the end, we simply decided to leverage the existing GetError semantics to capture this "sticky" error behavior.

One additional concern was that gl errors are traditionally only used to indicate programming errors on the part of the application, but the framebuffer completeness test may have failed simply because of implementation dependencies through no fault of the application. We decided to adopt the notion is that it is an error to attempt to render with an incomplete framebuffer, on all implementations, and so it actually *is* a programming error if an application does not attempt to deal with an incomplete framebuffer prior to rendering.

(65) *If it is an error to render to or read from an incomplete framebuffer, should we use INVALID_OPERATION or create a new error?*

RESOLUTION: resolved, INVALID_FRAMEBUFFER_OPERATION_EXT

We resolved to create a new error at the September ARB meeting and then resolved the name of the error within the work group.

> We agreed that if we throw an error here, we'd like a new error enum, particularly because the error may have been triggered by a framebuffer which is incomplete for implementation dependent reasons.

Some options for the new error name which were discussed:

    OPERATION_ON_INCOMPLETE_FRAMEBUFFER
    INCOMPLETE_FRAMEBUFFER
    IMPLEMENTATION_DEPENDENT_FAILURE
    INVALID_FRAMEBUFFER
    INVALID_FRAMEBUFFER_OPERATION

(66) There are several issues related to how we treat DrawBuffer(s) and other context state with respect to framebuffer completeness. We'd like a self-consistent model here and this may affect the resolution of issue (8), (44), (55), and (56).

RESOLUTION: resolved, (d) - no context state in framebuffer completeness test, but context state can affect whether rendering takes place, does not take place, or is undefined. Note option (d) required us to revisit issue (44).

The first question we had to answer was:

> Is it desireable that "framebuffer completeness" be purely a property of the set of framebuffer state (which includes the state of the images attached to the framebuffer)? Or can a framebuffer's completeness depend on "non-framebuffer" context state as well?

For instance, there are currently two pieces of context state that can affect framebuffer completeness: texture binding state and draw buffer state.

First, in issue (44), we decided that attaching an image of a currently bound and enabled texture to a framebuffer can cause a framebuffer to be incomplete. The texture binding is context state and there are pieces of the texture object state (base level, max level) that can also affect the determination of framebuffer completeness. (Additionally if we add render-to-vertex-array functionality later, we might expect to have a framebuffer completeness requirement that examines the state of the currently bound vertex array.)

One way to avoid this context dependency is to revisit issue (44) and say that this "texture-from-destination" case simply generates undefined rendering but does not affect framebuffer completeness. This would replace the "expressly disabled" rendering and framebuffer incompleteness with "undefined rendering", but would also let implementations avoid checking context state during the validation of the framebuffer state.

The second piece of context state that might cause framebuffer validation failures is the draw buffer(s) and/or read buffer state. It has been suggested in issue (55) that if the draw buffers specify attachment points with no attached images, then the framebuffer might be considered incomplete. If we choose to do this, then we would have context state influencing framebuffer completeness state. However, if we resolve issue (56) to say that the draw

```
        buffer state is part of the framebuffer object state, then
        the draw buffer is no longer context state and this
        particular dependency of framebuffer completeness on context
        state goes away.

        The above discussion leaves us with several self-consistent,
        but different sets of decisions:

            (a) Remove context dependencies from framebuffer
                completeness.

                To do this we would:

                - Move draw buffer state from context into
                  framebuffer: issue (56)

                - Make "texture-from-destination" undefined instead
                  of a reason for framebuffer incompleteness: issue
                  (44)

                - Presumably, if we created a render-to-vertex-array
                  extension layered on this one, we would likely
                  also make rendering into the currently bound
                  vertex array undefined as well.

                With option (a), we can say that having draw buffer
                set to an non-existent buffer is a reason for
                framebuffer incompleteness and there are no context
                dependencies.  This would resolve issue (55).

            (b) Allow context dependencies in framebuffer
                completeness.

                Essentially this means that the result of a query of
                framebuffer completeness is dependent on the context
                making the query - or put another way, the
                framebuffer completeness state is context state not
                framebuffer state.

                If we choose this option (b), then we are esentially
                free to resolve issues (44), (55), and (56)
                however we want.  In other words:

                    - draw buffer can be either context or
                      framebuffer state: issue (56)

                    - "texture-from-destination" can be either
                      undefined or a reason for framebuffer
                      incompleteness

                    - draw buffer specifying a non-existent buffer
                      can be a reason for framebuffer incompleteness
                      or could result in undefined behavior: issue
                      (55)
            (c) Remove the framebuffer object and make the
                framebuffer state part of the context.

                This option redefines the issue by not making a
                distinction between framebuffer "object" state and
                "context" state, therefore framebuffer completeness
                depends only on "context" state because all of the
                "framebuffer" state is now "context" state.

                This would mean that there is now a subset of state
                in the context that can be considered the
                "framebuffer state" of the context.  This is the set
                of state that would presumably be pushed/popped
                under a theoretical FRAMEBUFFER_BIT for
                PushAttrib().

                Regardless of whether there is a framebuffer object,
                framebuffer completeness may or may not still depend
                on pieces of other "context" state that are not part
                of subset of context state related to the
                "non-default" framebuffer (for instance, texture
                bindings and/or draw buffer state).

                If we choose this option (c),

                    - we remove the framebuffer object: issue (8)
                      This means:

                        - removing gen/is/bind/delete framebuffer
                          object

                        - moving the attachment state into the
                          context

                        - creating new context bind points for
                          framebuffer attachments and creating new
                          BindFramebufferAttachableImage calls or
                          using the FramebufferTexture() calls to do
                          context binds of framebuffer-attachable
                          images

                    - we decide whether there is a single set of
                      draw/read buffer context state or a 2nd set of
                      draw/read buffer context state to be used for
                      "non-default" framebuffer objects.  Either way
                      it's "context" state but we need to know if we
                      have one set of state or two.  This is a
                      variation on issue (56).

                    - as in option (b), "texture-from-destination"
                      can be either undefined or a reason for
                      framebuffer incompleteness

                    - as in option (b), a draw buffer specifying a
                      non-existent buffer can either be a reason for
                      framebuffer incompleteness or could result in
                      undefined behavior: issue (55)
                    - all the framebuffer attachments become context
                      state

                    - we add a framebuffer enable/disable bit to use
                      to distinguish between the "default" and
                      "non-default" framebuffer

            (d) Create a new category of reasons that you can't use
                a framebuffer for rendering in a specific context,
                but that are not part of the test for "framebuffer
                completeness"

                Essentially, this is a kind of hybrid of options (a)
                and (b).  There are no context dependent reasons for
                framebuffer incompleteness, but at the same time
                there are some additional context-dependent
                constraints on using a framebuffer.  In other words,
                a framebuffer can be complete but still not suitable
                for rendering by a given context.

                This creates two categories of tests that can be
                used to disable rendering - the set of
                context-independent test that are used to determine
                framebuffer completeness, and the set of tests that
                are context-dependent and not used to determine
                framebuffer completeness.

                An open question is: should we add a separate query
                for this second set of context-dependent tests
                and/or a "meta-query" that would cover both sets.
                This "meta-query" would return "true" if and only if
                the framebuffer is complete *and* it can be used in
                this context.

                Note that while the "is framebuffer complete" query
                is required by the fact that a framebuffer can be
                incomplete because of implementation dependent
                reasons, the second query of the context-dependent
                test results and the "meta query" are primarily
                debugging aids, though perhaps convenient ones.

                The framebuffer completeness query is analogous to
                asking if a texture is "mipmap complete".  The
                question, "can I render into my framebuffer", is
                analgous to asking the question, "is texturing
                enabled."  A bound texture may be "complete", but
                texturing can still be disabled due to an
                unfortunate combination of non-texture-object
                context state.  Option (d) is basically saying the
                same thing of framebuffer objects.

                To implement option (d), we'd do the following:

                - If draw buffer is defined as "context state" it
                  can not affect framebuffer completeness, but if
                  draw buffer is defined as framebuffer state it
```

might affect framebuffer completeness. See issues (55) and (56).

- Make "texture-from-destination" undefined instead of a reason for framebuffer incompleteness: issue (44). Technically, this could still be an error unrelated to framebuffer completeness, but we are trying to avoid creating a precedent for arbitrary "errors at begin time". When this case was included in the "framebuffer completeness" validation, the additional cost of generating the error was free. But if this "texture-from-destination" case is not part of framebuffer completeness, then it is an additional cost at begin time to detect this in order to flag an error (and/or disable rendering). To avoid this cost, we would make this undefined.

- Presumably, if we later create a render-to-vertex-array extension layered on this one, we would likely also choose the same resolution for rendering into the currently bound vertex array as we choose for the currently bound texture.

(67) In issue (63) we decided we want to use a dedicated API function to test framebuffer completeness. We might want to change the name of "ValidateFramebuffer" however. If so, what name should we use?

RESOLUTION: resolved, CheckFramebufferStatus()

One reason we decided to retain an explicit API function instead of just using a GetInteger style query is to emphasis the "on-demand" state examination that takes place when making this query.

However, some were uncomfortable with the name ValidateFramebuffer for this purpose. Some felt that it implied a requirement to call the function, and others felt it was too similar in name to the GLSL function ValidateProgram which served a related but slightly different purpose. So we chose a new name.

Some options we considered:

```
    ValidateFramebufferCompleteness()
    CheckFramebufferCompleteness()
    CheckFramebufferStatus()
    IsFramebufferComplete()
```

(68) Exactly which levels should by generated by GenerateMipmapEXT?

RESOLUTION: resolved, from TEXTURE_BASE_LEVEL+1 through q

Automatic mipmap generation via GENERATE_MIPMAP generates from TEXTURE_BASE_LEVEL+1 through p, which is the 1x1 level. However, applications frequently don't want to waste

        computation generating past q, which is the min of
        TEXTURE_MAX_LEVEL and p. The only recourse is to accept the
        performance hit or to not use GENERATE_MIPMAP.

        Arguably GENERATE_MIPMAP should have been specified to
        generate only through q. We have the opportunity to "fix"
        this problem by "correctly" specifying the new function
        GenerateMipmapEXT to generate only from TEXTURE_BASE_LEVEL+1
        through q.

        As the specification of GenerateMipmapEXT is currently
        written, GenerateMipmapEXT only generates levels
        TEXTURE_BASE_LEVEL+1 through q.

(69) What should we call the framebuffer objects to distinguish them from the default framebuffer?

        RESOLUTION: resolved, "application-created"

        Currently we call these "application-created" framebuffers
        Some places in the spec have also referred these as
        "GL-allocated" framebuffers. Whichever term we use, we
        should use it consistently.

        Some terms we considered:

            "application-created" framebuffers
            "application-allocated" framebuffers
            "non-default" framebuffers
            "GL-created" framebuffers
            "GL-allocated" framebuffers
            "dynamically-created" framebuffers
            etc.

        The GL spec already talks about "creating" textures, not
        "allocating" them, so "*-created" seems like a better
        choice.

        It's a bit of a toss-up between "GL-created" and
        "application-created". Technically, the "GL" really creates
        and manages these objects but it only does so at the request
        of the application. Going with "application-created" for
        now.

(70) With which, if any, attribute bit does the framebuffer binding push and pop? The same question applies to the current renderbuffer?

        RESOLUTION: resolved, don't push/pop framebuffer binding
        bit for now. If desired, we may add this in the ARB/core
        update of this spec.

    There are a few precedents to choose from.

    The ARB_vertex/fragment_program extensions chose to *not*
    push/pop the current program object binding. It's not clear if
    this was intentional or which existing attribute bit was

appropriate to use or if there was a desire to not create a new attribute bit.

ARB_vertex_buffer_object buffer objects and GL core texture objects do push/pop the bindings with the existing VERTEX_ARRAY and TEXTURE bits respectively. In addition, the texture enables are push/pop'ed with the TEXTURE bit.

If we do wish to push/pop the FRAMEBUFFER_BINDING_EXT state we probably need a new FRAMEBUFFER bit.

We could also consider adding a RENDERBUFFER_BIT to cover the current renderbuffer binding or allow this renderbuffer binding to push/pop with the FRAMEBUFFER bit. However, it's less clear that push/pop'ing the renderbuffer binding is useful since the renderbuffer binding is not used for rendering. The renderbuffer binding is only used to set the current renderbuffer for renderbuffer storage allocation and queries.

Also, there are a related set of questions about how much state should push/pop with a new FRAMEBUFFER bit. Should we push/pop all of the framebuffer object state in addition to the current binding? Similar to the way vertex array's can be attached to VBO's, use of VBO, framebuffers can be attached to other GL objects. The TEXTURE_BIT covers both per object (min/mag filter) and per context (texture environment and enable) state. It's not clear if this is useful or desirable to have per-object state push/pop. With the addition of object semantics, it seems like the need for push/pop of object state is reduced.

In the end, since we'd need to create a new bit anyway, we decided to defer adding push/pop semantics until we understand the implementation ramifications better. If we decide to create the bit later on in the ARB or Core revision of this extension, we can add it in a backward-compatible fashion.

(71) Should we spell out precisely which rendering and reading routines can cause us to generate an error at the time the rendering or reading functions are called?

RESOLUTION: resolved, keep the same language as ARB vertex/fragment program and GLSL for now, with the addtitions relevant for reading the framebuffer, but recommend the ARB look at this when doing the next core GL spec revision.

Currently GL has a few cases that can cause errors at render time. Specifically, attempting to render with a mapped vertex buffer object, an invalid low-level vertex or fragment program, or an invalid GLSL program object all generate errors at "Begin" time.

This extension adds a new error at "begin" time. Attempting to render with an "incomplete" framebuffer generates INVALID_FRAMEBUFFER_OPERATION_EXT. In addition, this extension adds the same error at "read" time if the application tries to read from an "incomplete" framebuffer.

```
    The ARB vertex program, ARB fragment program, and GLSL extension
    specs state that an app which tries to use an "invalid" object
    can generate errors when Begin, RasterPos, or any command that
    performs an explicit Begin is called.

    This extension has adopted similar language.  So the question
    asked by this issue is: do we need ot be more explicit.

    There are some ambiguities.  For instance, it is an error to
    write pixels using an "implicit Begin" operation like DrawArrays
    if the current vertex program is invalid, but it is not an error
    to do an Accum operation which also writes pixels to the
    framebuffer.

    This issue applies to all of these extensions.

    Options include:

        - listing all routines which can render or read from the
          framebuffer and stating that they can cause an error if
          the framebuffer is incomplete, solving the problem for
          this extension only.

        - adding to the GL core a table of "routines that read
          pixels" and "routines that write pixels" and referencing
          those tables in the language for each of these extensions.

    Because each extension is doing something a little different,
    it's not even clear if the second option is a viable option.
    It's possible each extension would need its own list of routines
    which can generate errors anyway.

    Basically, this is a larger problem than this
    EXT_framebuffer_object extension.  For now, we choose to use the
    same (vague-ish) language adopted by the
    ARB_vertex/fragment_program and GLSL extnesions.

    We do recommend, however, that the ARB address this issue in the
    next GL core revision.
```

(72) *Should the framebuffer completeness test include a clause that says "at least one color attachment" has been made? Or "at least one attachment of any type"? Or is the framebuffer still complete when there are no attachments at all?*

```
        RESOLUTION: resolved, a framebuffer must have at least one
        color-renderable, depth-renderable, or stencil-renderable
        image attached to be complete.

        While a framebuffer with only depth, or only color
        attachments seems plausible, we couldn't come up with a
        sensible use for a framebuffer with no attachments at all,
        so the assumption is that this is an unintended error on the
        part of the application.  Therefore, we choose to make it
        part of the framebuffer completeness test.
```

We could make this its own clause in the framebuffer completeness test. If we choose to do so, we should probably come up with a new FRAMEBUFFER_INCOMPLETE_* to conform to our previous practice of keeping one enum per clause.

However, since this is really related to the attachment state, we could just piggy back this on the first clause and same all the attachment points must be "attachment complete" and there must be at least one color, depth, or stencil buffer attached.

If we choose this latter option, we can continue to use the FRAMEBUFFER_INCOMPLETE_ATTACHMENT enum to cover this case.

(73) *This clause from framebuffer completeness (before it was reworded, see below):*

* The value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE_EXT must not be NONE for any color attachment point named by READ_BUFFER.

basically requires at least one color attachment is non-NULL. But this is not what we want. So what should we do?

RESOLUTION: resolved, (4a) READ_BUFFER can be NONE

The reason is: READ_BUFFER is not allowed to be NONE, which in turn means to be framebuffer complete, READ_BUFFER must be COLOR_ATTACHMENTn_EXT for some n which has an image attached. However, we don't wish to preclude a no-color framebuffer. What should we do?

Options include:

4a) Allow READ_BUFFER of NONE, reads of color from the framebuffer when read buffer is none, generate error INVALID_OPERATION

4b) Generate an error when a read operation (ReadPixels, CopyPixels, etc) is attempted while the color attachment point referenced by the READ_BUFFER does not have an attached image.

4c) Reverse earlier decision to allow complete framebuffer not to have any color attachments. Instead, require at least one color attachment. READ_BUFFER must point to a valid color attachment or else the framebuffer object is incomplete.

(4c) seems to require the user attach a color buffer just to be able to read the depth buffer of a depth-only framebuffer.

(4b) seems to suffer from the same problem (unless we move the "valid read buffer" test out of the completeness test).

Of these choices, (4a) seems to be the most palatable. We choose the allow the value of READ_BUFFER to be NONE, but reads of color buffers when READ_BUFFER is NONE will generate an error, in order to be consistent with the decision in issues (26) and (65).

Note: that clause was eventually reworded to say:

* If READ_BUFFER is not NONE, then the value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE_EXT must not be NONE for the color attachment point named by READ_BUFFER.

(74) What should CheckFramebufferStatusEXT return if FRAMEBUFFER_BINDING_EXT is zero?

Secondary question: what should CheckFramebufferStatusEXT return if there is an error?

RESOLUTION: resolved, default fb returns COMPLETE always, and CheckFramebufferStatusEXT returns FRAMEBUFFER_STATUS_ERROR if there is an error (bad target)

This goes to a larger question of whether all framebuffers including the default window-system-provided framebuffer have a "completeness" state, or if "completeness" is only a property which applies to application-created framebuffers.

For the case where the current FRAMEBUFFER_BINDING_EXT is zero, options include:

- CheckFramebufferStatusEXT returns an error when FRAMEBUFFER_BINDING_EXT is zero.

- CheckFramebufferStatusEXT always returns FRAMEBUFFER_COMPLETE_EXT when FRAMEBUFFER_BINDING_EXT is zero.

For the case CheckFramebufferStatusEXT generates an error, options include:

- reworking CheckFramebufferStatus into a "get" style routine that returns a value (or not) in an input parameter like GetIntegerv

- returning a known value like NONE or 0

- returning undefined results

(75) How are state values for the stencil index write mask and stencil reference value affected by this extension?

RESOLUTION:
a) index write mask is stored as 32 bit value, default is all 1's, and
b) reference value is not clamped on specification but rather is clamped on use and query, and
c) we need to add the stencil reference value

to the state table that lists the state values that might change after a framebuffer state change

The reason this is an issue is that the current GL specification indicates that the stencil index write mask and the stencil reference value are masked/clamped according to the number of stencil bitplanes. However, in this extension the number of stencil bitplanes can now change dynamically as the image attached to the framebuffer is changed.

For instance, if these values are clamped/masked according to the bitdepth of the currently attached stencil buffer, what should happen if the user later attaches a stencil buffer of a different bit depth? Must the stencil reference value or index write mask be respecified?

For the index write mask: we decide to treat this value as "all 1's" as the current specification allows, but further define the number of 1's to be 32 (the minimum width of an integer in GL), and a likely maximum stencil bitdpeth for the forseeable future. This should retain backward compatbility and still handle the case where the bitdepth of the stencil buffer can change dynamically.

For the stencil reference value, we decide to treat this state similar to way various clamped colors are treated in the ARB floating point pixel extensions. Specifically, the state values are clamped against the current logical buffer bitdepths as they are used for rendering and queried, but are not clamped on specification. This means that these state values do not need to be respecified just because the logical buffer bit depth changes, and retains backward compatibility to the behavior prior to this extension.

We will update the appropriate sections of the specification to describe this behavior.

(76) Currently framebuffer objects are shared, should we make them not shared across contexts?

RESOLUTION: yes, framebuffers are shared like display lists and textures are shared.

Initially it was suggested that some complicated multi-context semantics might be avoided if if the namespace for framebuffer objects were not shared across contexts. Specifically, some members of the group felt that by not sharing framebuffer objects, we could avoid the situation where:

a) one context can change the draw buffer of a framebuffer object in use by another context.

b) one context can change the attachments of a framebuffer object which may be in use by another context.

However, after some discussion, we realized that even if we didn't share framebuffer objects, there were still interactions similar to those listed above because the underlying images could still be shared. Consequently, one context could still affect the completeness and attachments of the framebuffers in another context by modifying or deleting the framebuffer-attachable images shared by both contexts.

So in the end, we decided to retain the share-ability of framebuffer objects rather than introduce an asymmetry with other GL objects like textures.

(77) If the application deletes an object and that object contains an image which is attached to a framebuffer object, exactly when and how is the image detached from the framebuffer?

RESOLUTION: resolved, option (1): images are detached from the currently bound framebuffer on delete, but images remain attached to any non-bound framebuffers.

This is issue is somewhat related to the multi-context object-sharing discussion currently going on in the ARB.

This extension presupposes that framebuffer attachments represent a reference to the attached image (or more correctly - a reference to the object containing the attached image). Since having a reference to an object affects when the object (and/or its name) is deleted, object deletion semantics are tied into the notion when the state describing these references is modified. In other words, the semantics of when objects are deleted are affected by the details concerning when a change to the framebuffer attachment state takes place.

Prior to the EXT_framebuffer_object and GLSL extensions, the only way in which an object not currently bound to this GL context could be modified, was when the object was modified by another GL context.

Both the EXT_framebuffer_object and GLSL extensions allow an object (texture, renderbuffer, shader) to be attached to a "container" object (framebuffer, program). With the introduction of "attachment", an object could be bound to the context at more than one binding point. For example, a texture can be bound to TEXTURE_2D_BINDING, and it can also be indirectly bound through the FRAMEBUFFER_BINDING if it is attached to the framebuffer object bound to the FRAMEBUFFER_BINDING.

Furthermore, a texture can be attached (by reference) to a framebuffer object that is not bound to any context, while at the same time the texture *is* bound to context's TEXTURE_2D_BINDING. Because the texture state is a part of the framebuffer object's state, it is now possible for modification of a texture through TEXTURE_2D_BINDING to cause modification of a framebuffer object, even though the framebuffer object is not bound to any context at the time it is modified.

One conceptual model for dealing with this situation is to treat attachment similar to bind, but instead of binding to a context, you are "binding" to another object. For the purposes of managing object references, object lifetimes, state propogation semantics, etc., these attachments can be considered to be "just like" a bind operation. [A "bind" and an "attach" are not exactly equivalent, however; see issue (82) for a further discussion on Bind vs. Attach.]

If we agree on the above conceptual model, then we may wish to look to the multi-context situation for guidance on how to treat state changes to non-currently-bound framebuffer objects.

Unfortunately, the multi-context semantics are poorly defined by OpenGL. If we decide to use them as a guide, we should at least define what they are and this is why the larger ARB is looking at this issue now.

For EXT_framebuffer_object, there are three choices for behavior. In each case, we defer to the larger ARB the details about when an object name is available for reuse. For the purposes of this discussion, we are looking only at state changes governing the attachments. The three choices are listed below:

For the sake of concrete simplicity, this discussion speaks to the images of a texture object; but it applies equally to the image of a renderbuffer object.

If you delete a texture object while one of the texture's images is attached to a framebuffer object (or multiple framebuffer objects), then:

(1) The image is automatically detached from the currently bound framebuffer object only.

If the image is also attached to any other framebuffer objects, then the image is NOT automatically detached from those.

The application is responsible for manually detaching images from the other framebuffer objects, by rebinding each framebuffer in turn and performing an explicit detach operation.

Until the application manually detaches the image from the other framebuffers, those framebuffers continue to use the image for rendering. The other framebuffer objects have a reference to the image until the image has been detached from them. In this way, attachment behaves as if the image was "bound to the framebuffer object".

(2) The image is automatically detached from the currently bound framebuffer object. Also during DeleteTexture, the image is automatically detached from any other framebuffer object to which it is attached; however, the image is not guaranteed to be detached from the other framebuffer objects until the next time those framebuffer objects are bound via BindFramebufferEXT.

Similar to option (1), in order to "really" delete the object, the application is responsible for rebinding all the framebuffer objects to which the deleted image was attached. However, unlike option (1), the application need not actually perform an explicit detach operation. The application can merely bind the framebuffer.

Until the application actually rebinds the framebuffer the images are not actually detached and deleted. The other framebuffer objects continue to hold a reference (like a binding) to the image until the next time the framebuffer objects are bound.

(3) The image is automatically detached from all framebuffers objects during DeleteTextures, including the currently bound framebuffer as well as any other framebuffers to which the image is attached.

The application need not explicitly bind to, and detach the image from, any framebuffer that is not bound at the time DeleteTextures was called.

Because the framebuffer object has a reference to the texture object, and the texture object's state is considered part of the framebuffer object's state, this resolution implies that DeleteTextures may modifiy the state of a framebuffer object that is not the currently bound object.

With reference to the object-sharing discussion that is going on in the ARB right now, for (a)-style implementations, options (2) and (3) are indistinguishable. However, for (b)-style implementations, implementing (3) would require textures to store a list of all attached framebuffers while (2) would not.

Options (2) and (3) essentially treat the currently-bound and non-currently-bound framebuffers the same--i.e., deleting the image (ultimately) detaches it from all framebuffer. This may be desirable as a convenience to the application.

On the other hand, Option (1) treats the currently bound framebuffer special, in that deletions are performed automatically much like textures are unbound automatically from the current context's binding points, but they are not unbound automatically from other contexts' binding points.

Also, Option (1) leaves the application in control of when the images are detached, which also may be desirable.

We choose option (1) because it is the simplest, and it also does not unduly burden implementations regardless of their choice of (a) versus (b) object-sharing model.

If an implementation has the (a)-style object sharing model, then the fact that images remain attached to non-bound objects has no affect on when the object name may be re-used. If the implementation has a (b)-style object-sharing model, then the outstanding attachments will delay re-use of the object name until the image has been detached. Regardless of whether the ARB chooses (a) or (b) behavior, or even if the ARB chooses to leave this behavior undefined, we can "piggy-back" on the name-reuse semantics they decide.

Also, option (1) means that if the application deletes a texture while one of the texture's images is attached to a framebuffer object that is not bound, then the application may continue to render into the image after the framebuffer is bound again, regardless of the (a) vs. (b) choice.

Finally, note that if a context deletes an object containing an image attached to the currently bound framebuffer, then we first detach the image from the bound framebuffer. This means that the state change to the framebuffer (the detach operation) is guaranteed to be picked up by any other context the next time the framebuffer is bound in one of the other contexts.

(78) Should we collapse the notions of "framebuffer-attachable image completeness" and "framebuffer attachment completeness" into a single type of completeness (probably retaining the name "framebuffer attachment completeness"

RESOLUTION: resolved, yes, eliminate "framebuffer-attachable image completeness" and add a "non-zero-area" requirement to the "framebuffer attachement completeness" test.

Originally this extension had several layers of which affected framebuffer completness. They were:

- framebuffer-attachable image completeness
    * image has non-zero width/height/depth
    * image has color, depth, or stencil format
    * image is not from a proxy texture

- framebuffer attachment completeness
    * attached image is textures/renderbuffer
    * attached image is from existing object
    * attached image has format appropriate for attachment point (depth buffer has depth format, etc)

```
            - framebuffer completeness
                * all attachment points are "attachment complete"
                * all images are "framebuffer-attachable image complete"
                * all color buffers have same format
                * draw buffer is attached
                * read buffer is attached
                * framebuffer format combination is supported

        However, upon further reflection of the
        "framebuffer-attachable image completeness" tests, we
        realized that

            a) the requirement that the renderble image is not a
               "proxy" texture was already covered by the fact that
               it's illegal to attach a proxy texture to a
               framebuffer, and

            b) the requirement that the format be color, depth, or
               stencil is essentially already covered by the
               "framebuffer attachment completeness" test
               requirement that the format is appropriate for the
               attachment point.

        This left only the "non-zero-area" test, so we decided to
        fold this requirement into the "framebuffer attachement
        completeness" test and eliminate the concept of
        "framebuffer-attachable image completeness".  This decision
        required the elimination of one of the
        FRAMEBUFFER_INCOMPLETE_* enums as they correspond to the
        conditions in the "framebuffer completeness" test of section
        4.4.4
```

(79) Should the internal format chosen by GL for a texture (or renderbuffer) be invariant with respect to the state of the current framebuffer and its attached images?

```
        RESOLUTION: yes, the choice of internal format must be
        invariant with respect to framebuffer state changes.

        This means that the GL must choose texture internal format
        based only on the arguments to TexImage and ignore the
        current framebuffer state in this selection process.

        Similarly, the GL must choose renderbuffer internal format
        based only on the arguments to RenderbufferStorage and
        ignore the current framebuffer state in this selection
        process.

        This issue is a variant of issue (62).  The OpenGL 2.0
        specification (p.152, paragraph 4) states that:

            "A GL implementation may vary its allocation of
            internal component resolution or compressed internal
            format based on any TexImage3D, TexImage2D (see below),
            or TexImage1D (see below) parameter (except target),
            but the allocation and chosen compressed image format
```

must not be a function of any other state and cannot be
changed once they are established."

Consider that prior to this extension, some implementations may have considered the the bitdepths of the logical buffers of the framebuffer or the bitdepth of the display when choosing an internal format for textures. Since, in practice, these bitdepths typically were immutable for the lifetime of a GL context, the invariance requirements were met.

With the introduction of EXT_framebuffer_object, however, the logical buffer bitdepths can change over the lifetime of the context. So this issue examines whether or not framebuffer state is allowed to affect texture internal format selection.

After some discussion, we felt it was too problematic to introduce this type of invariance. So this extension makes no modifications to the invariance language, and adds similar invariance language applicable to renderbuffer objects. As long as the app provides the same arguments to TexImage{1D|2D|3D} or RenderbufferStorage, then the GL must always choose the same internal format.

Note, however, that the GL is not required to provide the same internal format resolution for renderbuffers as it does for textures.

(80) Should attachment routines be display-list'able?

RESOLUTION: no, in fact none of the routines introduced in this extension are included in display lists.

Initially, we were just considering whether or not the framebuffer attachement routines should be included in display lists. The rationale for not including them was that since query routines can not be in display lists, and well-behaved apps should call the query routine CheckFramebufferStatusEXT() after calling making changes to framebuffer attachments, it was not possible to write a well-behaved app that uses display lists to build up and use a framebuffer. So, one possible solution was to simply disallow from display lists the routines that change change the result of CheckFramebufferStatusEXT().

However, we realized on further consideration that other routines which can affect the results of CheckFramebufferStatusEXT are already allowed in display lists. Namely, the routines which affect textures (TexImage and friends). So, disallowing the attachment routines is a partial solution at best.

We also looked for various precedents and found some mixed results:

- VBO bind operations are not display-list'able, but this is primarily because the VBO bindings are considered client-state

- texture bindings are display-list'able

In the end, though we decided to not include the routines introduced by this extension in display lists for reasons of simplicity more than anything else.

It's possible we may need to add support for display lists back in during promotion of this extension if we determine that it is needed later, but for now, we leave this out.

(81) How should PushAttrib and PopAttrib work with this extension?

RESOLUTION: mostly deferred for now, may revisit later

This extension introduces no new push/pop attrib bits to cover the state introduced by this extension (for instance there is no FRAMEBUFFER_BIT). So the only real question to answer is what effect should Push/Pop attrib have on any existing state as it relates to this extension.

In particular, how should Push/PopAttrib of the COLOR_BUFFER_BIT which covers the DRAW_BUFFER and READ_BUFFER state interact with this extension? Does the COLOR_BUFFER_BIT affect the per-object DRAW_BUFFER and READ_BUFFER state?

Currently, the answer is yes. PushAttrib(COLOR_BUFFER_BIT) saves the DRAW_BUFFER value of the currently bound framebuffer object. If the app later calls PopAttrib() this saved value will be restored even if the framebuffer bound at the time PopAttrib is called is different from the framebuffer bound at the time PushAttrib was called.

In other words, one are considering whether or not it is strange that PushAttrib(COLOR_BUFFER_BIT) affects a piece of per-object state. Note that this is somewhat similar to the way that a PushAttrib(TEXTURE_BIT) can save off per-texture-object state and a later call to PopAttrib can restore that per-object state even if the texture bound at PopAttrib time has since been changed/ deleted/modified in some way.

There are some differences with the texture analogy though. Namely, the TEXTURE_BIT does include the texture bindings so at least the texture object binding is restored in conjunction with the per-texture-object state. Also, some may consider the fact that the TEXTURE_BIT affects per-texture-object state more intuitive than the fact that the COLOR_BUFFER_BIT affects per-framebuffer-object state.

Also, there is a larger discussion going on in the ARB right now about whether PushAttrib/PopAttrib save references to existing bound objects or only the state values which name an existing bound object.

For now, we have deferred further discussion of the PushAttrib/PopAttrib semantics in this extension until the larger issues are cleared up.

(82) What is the relationship between a "binding" and an "attachment"?

RESOLUTION: resolved, the concept of attachment is described below, though final implications will be affected by larger ARB discussions about object sharing and multiple context semantics.

> "Attaching" is the act of connecting one object to another object.
>
> An "attach" operation is similar to a "bind" operation in that both represent a reference to the attached or bound object for the purpose of managing object lifetimes and both enable manipulation of the state of the attached or bound object.
>
> However, an "attach" is also different from a "bind" in that "binding" an unused object creates a new object, while "attaching" does not. Additionally, "bind" establishes a connection between a context and an object, while "attach" establishes a connection between two objects.
>
> Finally, if object "A" is attached to object "B" and object "B" is bound to context "C", then in principle, we treat "A" as if it is <implicitly> bound to "C".
>
> The larger ARB is currently attempting to more clearly define the mutliple context semantics as they relate to object sharing and binding. The final implications for EXT_framebuffer_object may not be clear until those discussions are resolved. This extension may need an update once those issues are addressed.

(83) We use a non-zero framebuffer binding to enable the use of this extension. Should we instead consider using an explicit enable?

RESOLVED: no, retain the "non-zero-binding means enable" semantics.

Currently we enable the use of an application-created framebuffer by binding a non-zero framebuffer object to FRAMEBUFFER_EXT binding point. If the framebuffer binding is zero, then the extension is disabled (i.e., we use the window-system-provided framebuffer).

It might be cleaner to be able to say things like, "when FRAMEBUFFER_OBJECT is enabled", rather than "when the framebuffer binding is not zero", and add an explicit enable. Doing so would also allow changing framebuffer object attachments while FBO is disabled, which might result in the driver doing less validation while the application is setting up framebuffer objects. It would also provide a cleaner way to explain that the permitted DRAW_BUFFER and READ_BUFFER values change when the extension is enabled/disabled.

There are a few object model precedents to choose from: Textures and ARB Vertex and Framgment Program extensions use the explicit enable state. However, Vertex Buffer Objects, Pixel Buffer Objects, and GLSL Vertex and Fragment shaders use a non-zero-binding to enable the use of those features.

If we used an explicit enable, then we could allow creation of an object named zero. Precedent dictates that an object named zero is never shared in the context share group. All other framebuffer objects are shared across the share group. It might be cleaner to disallow creation of an object named zero anyway.

Since binding to zero disables the extension, one way to think about this is that there is an object named zero which is managed through MakeCurrent, MakeContextCurrent, and the window manager. All other objects are managed through FramebufferTexture, FramebufferRenderbuffer, and the operations that define/modify texture and renderbuffer images. When looked at in this light, lack of an explicit enable is not as strange.

(84) Do we need to add any language to describe the y-orientation of framebuffer-attachable images? Specifically, what coordinate system is used by images attached to the framebuffer?

Resolution: unresolved

GL defines the rendering origin at the lower-left corner. Yet, because of the differences between orientation storage of textures and images, pbuffer rendering is often implemented using a "y-inverted" coordinate system. Is this y-inversion exposed in the API?

Is the origin in the lower-left? Upper-left? Do we need to say anything about this at all, or is it already covered by existing GL language?

Currently there is place-holder text in section 4.4.2.3

The intent of this specification is simply to mirror the y-orientation issues of the pbuffer style render to texture API's. What's unclear is whether this requires any new language in this specification or not.

(85) Explain what happens when the FBO has a different width/height from the window?

An FBO takes on the width/height of its attachments. This width/height can be different than the width/height of the window (of framebuffer "zero").

In such cases, after calling BindFramebuffer, it is the application's responsibility to set the glViewport and the glScissor (if necessary) to match that of the new framebuffer binding.

Some background: The default viewport for a context is defined by the dimensions of the window to which the context is first made current. When the window is resized, or when the context is bound to a different window, the viewport does not change automatically. It has always been the application's responsibility to set the viewport after one of these events. FBO is no different; after BindFramebuffer it is the application's responsibility to set the viewport if the new framebuffer binding has a different width/height than the old binding..

(86) Are any one- or two- component formats color-renderable?

Presently none of the one- or two- component texture formats defined in unextended OpenGL is color-renderable. The R and RG float formats defined by the NV_float_buffer extension are color-renderable.

Although an early draft of the FBO specification permitted rendering into alpha, luminance, and intensity formats, this this capability was pulled when it was realized that it was under-specified exactly how rendering into these formats would work. (specifically, how R/G/B/A map to I/L/A)

To resolve this we seem to have two options:

A) Add new R and RG formats like NV_float_buffer did.

B) For the existing one- and two- component formats, define the mapping from RGBA components to ILA components.

The superbuffers group has informally decided that option A is preferable.

(87) What happens if a single image is attached more than once to a framebuffer object?

RESOLVED: The value written to the pixel is undefined.

There used to be a rule in section 4.4.4.2 that resulted in FRAMEBUFFER_INCOMPLETE_DUPLICATE_ATTACHMENT_EXT if a single image was attached more than once to a framebuffer object.

```
    FRAMEBUFFER_INCOMPLETE_DUPLICATE_ATTACHMENT_EXT   0x8CD8
```

```
            * A single image is not attached more than once to the
              framebuffer object.

              { FRAMEBUFFER_INCOMPLETE_DUPLICATE_ATTACHMENT_EXT }

        This rule was removed in version #117 of the
        EXT_framebuffer_object specification after discussion at the
        September 2005 ARB meeting.  The rule essentially required an
        O(n*lg(n)) search.  Some implementations would not need to do that
        search if the completeness rules did not require it.  Instead,
        language was added to section 4.10 which says the values
        written to the framebuffer are undefined when this rule is
        violated.
```

Revision History

    abridged

Name

    EXT_framebuffer_sRGB

Name Strings

    GL_EXT_framebuffer_sRGB
    GLX_EXT_framebuffer_sRGB
    WGL_EXT_framebuffer_sRGB

Contributors

    Herb (Charles) Kuta, Quantum3D

    From the EXT_texture_sRGB specification...

    Alain Bouchard, Matrox
    Brian Paul, Tungsten Graphics
    Daniel Vogel, Epic Games
    Eric Werness, NVIDIA
    Kiril Vidimce, Pixar
    Mark J. Kilgard, NVIDIA
    Pat Brown, NVIDIA
    Yanjun Zhang, S3 Graphics
    Jeremy Sandmel, Apple

Contact

    Mark J. Kilgard, NVIDIA Corporation (mjk 'at' nvidia.com)

Status

    Shipping for GeForce 8 Series (November 2006)

Version

    Date: November 6, 2006
    Revision: 2

Number

    337

Dependencies

    OpenGL 1.1 required

    This extension is written against the OpenGL 2.0 (September 7,
    2004) specification.

    WGL_EXT_extensions_string is required for WGL support.

    WGL_EXT_pixel_format is required for WGL support.

    ARB_color_buffer_float interacts with this extension.

    EXT_framebuffer_object interacts with this extension.

EXT_texture_sRGB interacts with this extension.

ARB_draw_buffers interacts with this extension.

Overview

Conventionally, OpenGL assumes framebuffer color components are stored in a linear color space. In particular, framebuffer blending is a linear operation.

The sRGB color space is based on typical (non-linear) monitor characteristics expected in a dimly lit office. It has been standardized by the International Electrotechnical Commission (IEC) as IEC 61966-2-1. The sRGB color space roughly corresponds to 2.2 gamma correction.

This extension adds a framebuffer capability for sRGB framebuffer update and blending. When blending is disabled but the new sRGB updated mode is enabled (assume the framebuffer supports the capability), high-precision linear color component values for red, green, and blue generated by fragment coloring are encoded for sRGB prior to being written into the framebuffer. When blending is enabled along with the new sRGB update mode, red, green, and blue framebuffer color components are treated as sRGB values that are converted to linear color values, blended with the high-precision color values generated by fragment coloring, and then the blend result is encoded for sRGB just prior to being written into the framebuffer.

The primary motivation for this extension is that it allows OpenGL applications to render into a framebuffer that is scanned to a monitor configured to assume framebuffer color values are sRGB encoded. This assumption is roughly true of most PC monitors with default gamma correction. This allows applications to achieve faithful color reproduction for OpenGL rendering without adjusting the monitor's gamma correction.

New Procedures and Functions

None

New Tokens

Accepted by the <attribList> parameter of glXChooseVisual, and by the <attrib> parameter of glXGetConfig:

```
    GLX_FRAMEBUFFER_SRGB_CAPABLE_EXT             0x20B2
```

Accepted by the <piAttributes> parameter of wglGetPixelFormatAttribivEXT, wglGetPixelFormatAttribfvEXT, and the <piAttribIList> and <pfAttribIList> of wglChoosePixelFormatEXT:

```
    WGL_FRAMEBUFFER_SRGB_CAPABLE_EXT             0x20A9
    Accepted by the <cap> parameter of Enable, Disable, and IsEnabled,
    and by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv,
    and GetDoublev:

        FRAMEBUFFER_SRGB_EXT                        0x8DB9

    Accepted by the <pname> parameter of GetBooleanv, GetIntegerv,
    GetFloatv, and GetDoublev:

        FRAMEBUFFER_SRGB_CAPABLE_EXT                0x8DBA
```

Additions to Chapter 2 of the 2.0 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 2.0 Specification (Rasterization)

None

Additions to Chapter 4 of the 2.0 Specification (Per-Fragment Operations and the Frame Buffer)

DELETE the following sentence from section 4.1.8 (Blending) because it is moved to the new "sRGB Conversion" section:

"Each of these floating-point values is clamped to [0,1] and converted back to a fixed-point value in the manner described in section 2.14.9."

If ARB_color_buffer_float is supported, the following paragraph is modified to eliminate the fixed-point clamping and conversion because this behavior is moved to the new "sRGB Conversion" section.

"If the color buffer is fixed-point, the components of the source and destination values and blend factors are clamped to [0, 1] prior to evaluating the blend equation, the components of the blending result are clamped to [0,1] and converted to fixed-point values in the manner described in section 2.14.9. If the color buffer is floating-point, no clamping occurs. The resulting four values are sent to the next operation."

The modified ARB_color_buffer_float paragraph should read:

"If the color buffer is fixed-point, the components of the source and destination values and blend factors are clamped to [0, 1] prior to evaluating the blend equation. If the color buffer is floating-point, no clamping occurs. The resulting four values are sent to the next operation."

```
    Replace the following sentence:

    "Destination (framebuffer) components are taken to be fixed-point
    values represented according to the scheme in section 2.14.9 (Final
    Color Processing), as are source (fragment) components."

    with the following sentences:

    "Destination (framebuffer) components are taken to be fixed-point
    values represented according to the scheme in section 2.14.9 (Final
    Color Processing).  If FRAMEBUFFER_SRGB_EXT is enabled and the boolean
    FRAMEBUFFER_SRGB_CAPABLE_EXT state for the drawable is true, the R,
    G, and B destination color values (after conversion from fixed-point
    to floating-point) are considered to be encoded for the sRGB color
    space and hence need to be linearized prior to their use in blending.
    Each R, G, and B component is linearized by some approximation of
    the following:

            {  cs / 12.92,                 cs <= 0.04045
       cl = {
            {  ((cs + 0.055)/1.055)^2.4,   cs >  0.04045

    where cs is the component value prior to linearization and cl is
    the result.  Otherwise if FRAMEBUFFER_SRGB_EXT is disabled, or the
    drawable is not sRGB capable, or the value corresponds to the A
    component, then cs = cl for such components.  The corresponding cs
    values for R, G, B, and A are recombined as the destination color
    used subsequently by blending."

    ADD new section 4.1.X "sRGB Conversion" after section 4.1.8 (Blending)
    and before section 4.1.9 (Dithering).  With this new section added,
    understand the "next operation" referred to in the section 4.1.8
    (Blending) to now be "sRGB Conversion" (instead of "Dithering").

    "If FRAMEBUFFER_SRGB_EXT is enabled and the boolean
    FRAMEBUFFER_SRGB_CAPABLE_EXT state for the drawable is true, the R,
    G, and B values after blending are converted into the non-linear
    sRGB color space by some approximation of the following:

             {  0.0,                          0         <= cl
             {  12.92 * c,                    0         <  cl < 0.0031308
        cs = {  1.055 * cl^0.41666 - 0.055,   0.0031308 <= cl < 1
             {  1.0,                                       cl >= 1

    where cl is the R, G, or B element and cs is the result
    (effectively converted into an sRGB color space).  Otherwise if
    FRAMEBUFFER_SRGB_EXT is disabled, or the drawable is not sRGB
    capable, or the value corresponds to the A element, then cs = cl
    for such elements.

    The resulting cs values form a new RGBA color value.  If the color
    buffer is fixed-point, the components of this RGBA color value are
    clamped to [0,1] and then converted to a fixed-point value in the
    manner described in section 2.14.9.  The resulting four values are
    sent to the subsequent dithering operation."
```

Additions to Chapter 5 of the 2.0 Specification (Special Functions)

None

Additions to Chapter 6 of the 2.0 Specification (State and State Requests)

None

Additions to the OpenGL Shading Language specification

None

Additions to the GLX Specification

None

Dependencies on ARB_color_buffer_float

If ARB_color_buffer_float is not supported, ignore the edits to ARB_color_buffer_float language.

Dependencies on EXT_texture_sRGB and EXT_framebuffer_object

If EXT_texture_sRGB and EXT_framebuffer_object are both supported, the implementation should set FRAMEBUFFER_SRGB_CAPABLE_EXT to false when rendering to a color texture that is not one of the EXT_texture_sRGB introduced internal formats. An implementation can determine whether or not it will set FRAMEBUFFER_SRGB_CAPABLE_EXT to true for the EXT_texture_sRGB introduced internal formats. Implementations are encouraged to allow sRGB update and blending when rendering to sRGB textures using EXT_framebuffer_object but this is not required. In any case, FRAMEBUFFER_SRGB_CAPABLE_EXT should indicate whether or not sRGB update and blending is supported.

Dependencies on ARB_draw_buffers, EXT_texture_sRGB, and EXT_framebuffer_object

If ARB_draw_buffers, EXT_texture_sRGB, and EXT_framebuffer_object are supported and an application attempts to render to a set of color buffers where some but not all of the color buffers are FRAMEBUFFER_SRGB_CAPABLE_EXT individually, the query of FRAMEBUFFER_SRGB_CAPABLE_EXT should return true.

However sRGB update and blending only apply to the color buffers that are actually sRGB-capable.

GLX Protocol

None.

Errors

Relaxation of INVALID_ENUM errors

Enable, Disable, IsEnabled, GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev now accept the new token as allowed in the "New Tokens" section.

New State

    Add to table 6.20 (Pixel Operations)

Get Value	Type	Get Command	Initial Value	Description	Sec.	Attribute
FRAMEBUFFER_SRGB_EXT	B	IsEnabled	False	sRGB update and blending enable	4.1.X	color-buffer/enable

    Add to table 6.33 (Implementation Dependent Values)

Get Value	Type	Get Command	Initial Value	Description	Sec.	Attribute
FRAMEBUFFER_SRGB_CAPABLE_EXT	B	IsEnabled	-	true if drawable supports sRGB update and blending	4.1.X	-

New Implementation Dependent State

    None

Issues

    1) *What should this extension be called?*

        RESOLVED: EXT_framebuffer_sRGB.

        The "EXT_framebuffer" part indicates the extension is in
        the framebuffer domain and "sRGB" indicates the extension is
        adding a set of sRGB formats. This mimics the naming of the
        EXT_texture_sRGB extension that adds sRGB texture formats.

        The mixed-case spelling of sRGB is the established usage so
        "_sRGB" is preferred to "_srgb". The "s" stands for standard
        (color space).

        For token names, we use "SRGB" since token names are uniformly
        capitalized.

    2) *Should alpha be sRGB encoded?*

        RESOLVED: No. Alpha remains linear.

        A rationale for this resolution is found in Alvy Ray's "Should
        Alpha Be Nonlinear If RGB Is?" Tech Memo 17 (December 14, 1998).
        See: ftp://ftp.alvyray.com/Acrobat/17_Nonln.pdf

    3) *Should the ability to support sRGB framebuffer update and blending
        be an attribute of the framebuffer?*

        RESOLVED: Yes. It should be a capability of some pixel formats
        (mostly likely just RGB8 and RGBA8) that says sRGB blending can
        be enabled.

        This allows an implementation to simply mark the existing RGB8
        and RGBA8 pixel formats as supporting sRGB blending and then

    just provide the functionality for sRGB update and blending for
    such formats.

    sRGB support for floating-point formats makes little sense
    (because floating-point already provide a non-linear distribution
    of precision and typically have considerably more precision
    than 8-bit fixed-point framebuffer components allow) and would
    be expensive to support.

    Requiring sRGB support for all fixed-point buffers means that
    support for 16-bit components or very small 5-bit or 6-bit
    components would require special sRGB conversion hardware.
    Typically sRGB is well-suited for 8-bit fixed-point components
    so we do not want this extension to require expensive tables
    for other component sizes that are unlikely to ever be used.
    Implementations could support sRGB conversion for any color
    framebuffer format but implementations are not required to
    (honestly nor are implementations like to support sRGB on anything
    but 8-bit fixed-point color formats).

4) *Should there be an enable for sRGB update and blending?*

    RESOLVED: Yes, and it is disabled by default. The enable only
    applies if the framebuffer's underlying pixel format is capable
    of sRGB update and blending. Otherwise, the enable is silently
    ignored (similar to how the multisample enables are ignored when
    the pixel format lacks multisample supports).

5) How is sRGB blending done?

    RESOLVED: Blending is a linear operation so should be performed
    on values in linear spaces. sRGB-encoded values are in a
    non-linear space so sRGB blending should convert sRGB-encoded
    values from the framebuffer to linear values, blend, and then
    sRGB-encode the result to store it in the framebuffer.

    The destination color RGB components are each converted
    from sRGB to a linear value. Blending is then performed.
    The source color and constant color are simply assumed to be
    treated as linear color components. Then the result of blending
    is converted to an sRGB encoding and stored in the framebuffer.

6) *What happens if GL_FRAMEBUFFER_SRGB_EXT is enabled (and
    GL_FRAMEBUFFER_SRGB_CAPABLE_EXT is true for the drawable) but
    GL_BLEND is not enabled?*

    RESOLVED: The color result from fragment coloring (the source
    color) is converted to an sRGB encoding and stored in the
    framebuffer.

7) *How are multiple render targets handled?*

    RESOLVED: Render targets that are not
    GL_FRAMEBUFFER_SRGB_CAPABLE_EXT ignore the state of the
    GL_FRAMEBUFFER_SRGB_EXT enable for sRGB update and blending.
    So only the render targets that are sRGB-capable perform sRGB
    blending and update when GL_FRAMEBUFFER_SRGB_EXT is enabled.

```
    8)  Should sRGB framebuffer support affect the pixel path?

        RESOLVED:  No.

        sRGB conversion only applies to color reads for blending and
        color writes.  Color reads for glReadPixels, glCopyPixels,
        or glAccum have no sRGB conversion applied.

        For pixel path operations, an application could use pixel maps
        or color tables to perform an sRGB-to-linear conversion with
        these lookup tables.

    9)  Can luminance (single color component) framebuffer formats
        support sRGB blending?

        RESOLVED:  Yes, if an implementation chooses to advertise such
        a format and set the sRGB attribute for the format too.

        Implementations are not obliged to provide such formats.

    10) Should all component sizes be supported for sRGB components or
        just 8-bit?

        RESOLVED:  This is at the implementation's discretion since
        the implementation decides what pixel formats such support sRGB
        update and blending.

        It likely implementations will only provide sRGB-capable
        framebuffer configurations for configurations with 8-bit
        components.

    11) What must be specified as far as how do you convert to and from
        sRGB and linear RGB color spaces?

        RESOLVED:  The specification language needs to only supply the
        linear RGB to sRGB conversion (see section 4.9.X above).

        The sRGB to linear RGB conversion is documented in the
        EXT_texture_sRGB specification.

        For completeness, the accepted linear RGB to sRGB conversion
        (the inverse of the function specified in section 3.8.x) is as
        follows:
```

Given a linear RGB component, cl, convert it to an sRGB component, cs, in the range [0,1], with this pseudo-code:

```
if (isnan(cl)) {
    /* Map IEEE-754 Not-a-number to zero. */
    cs = 0.0;
} else if (cl > 1.0) {
    cs = 1.0;
} else if (cl < 0.0) {
    cs = 0.0;
} else if (cl < 0.0031308) {
    cs = 12.92 * cl;
} else {
    cs = 1.055 * pow(cl, 0.41666) - 0.055;
}
```

The NaN behavior in the pseudo-code is recommended but not specified in the actual specification language.

sRGB components are typically stored as unsigned 8-bit fixed-point values. If cs is computed with the above pseudo-code, cs can be converted to a [0,255] integer with this formula:

```
csi = floor(255.0 * cs + 0.5)
```

12) *Does this extension guarantee images rendered with sRGB textures will "look good" when output to a device supporting an sRGB color space?*

RESOLVED: No.

Whether the displayed framebuffer is displayed to a monitor that faithfully reproduces the sRGB color space is beyond the scope of this extension. This involves the gamma correction and color calibration of the physical display device.

13) *How does this extension interact with EXT_framebuffer_object?*

RESOLVED: When rendering to a color texture, an application can query GL_FRAMEBUFFER_SRGB_CAPABLE_EXT to determine if the color texture image is capable of sRGB rendering.

This boolean should be false for all texture internal formats except may be true (but are not required to be true) for the sRGB internal formats introduced by EXT_texture_sRGB. The expectation is that implementations of this extension will be able to support sRGB update and blending of sRGB textures.

14) *How is the constant blend color handled for sRGB framebuffers?*

RESOLVED: The constant blend color is specified as four floating-point values. Given that the texture border color can be specified at such high precision, it is always treated as a linear RGBA value.

15) *How does glCopyTex[Sub]Image work with sRGB? Suppose we're rendering to a floating point pbuffer or framebuffer object and do CopyTexImage. Are the linear framebuffer values converted to sRGB during the copy?*

RESOLVED: No, linear framebuffer values will NOT be automatically converted to the sRGB encoding during the copy. If such a conversion is desired, as explained in issue 12, the red, green, and blue pixel map functionality can be used to implement a linear-to-sRGB encoding translation.

16) *Should this extension explicitly specify the particular sRGB-to-linear and linear-to-sRGB conversions it uses?*

RESOLVED: The conversions are explicitly specified but allowance for approximations is provided. The expectation is that the implementation is likely to use a table to implement the conversions the conversion is necessarily then an approximation.

17) How does this extension interact with multisampling?

RESOLVED: There are no explicit interactions. However, arguably if the color samples for multisampling are sRGB encoded, the samples should be linearized before being "resolved" for display and then recoverted to sRGB if the output device expects sRGB encoded color components.

This is really a video scan-out issue and beyond the scope of this extension which is focused on the rendering issues. However some implementation advice is provided:

The implementation sufficiently aware of the gamma correction configured for the display device could decide to perform an sRGB-correct multisample resolve. Whether this occurs or not could be determined by a control panel setting or inferred by the application's use of this extension.

18) Why is the sRGB framebuffer GL_FRAMEBUFFER_SRGB_EXT enable disabled by default?

RESOLVED: This extension could have a boolean sRGB-versus-non-sRGB pixel format configuration mode that determined whether or not sRGB framebuffer update and blending occurs. The problem with this approach is 1) it creates may more pixel formation configurations because sRGB and non-sRGB versions of lots of existing configurations must be advertised, and 2) applicaitons unaware of sRGB might unknowingly select an sRGB configuration and then generate over-bright rendering.

It seems more appropriate to have a capability for sRGB framebuffer update and blending that is disabled by default. This allows existing RGB8 and RGBA8 framebuffer configurations to be marked as sRGB capable (so no additional configurations need be enumerated). Applications that desire sRGB rendering should identify an sRGB-capable framebuffer configuration and then enable sRGB rendering.

This is different from how EXT_texture_sRGB handles sRGB support for texture formats. In the EXT_texture_sRGB extension, textures are either sRGB or non-sRGB and there is no texture parameter to switch textures between the two modes. This makes sense for EXT_texture_sRGB because it allows implementations to fake sRGB textures with higher-precision linear textures that simply convert sRGB-encoded texels to sufficiently precise linear RGB values.

Texture formats also don't have the problem enumerated pixel format descriptions have where a naive application could stumble upon an sRGB-capable pixel format. sRGB textures require explicit use of one of the new EXT_texture_sRGB-introduced internal formats.

19) How does sRGB and this extension interact with digital video output standards, in particular DVI?

RESOLVED: The DVI 1.0 specification recommends "as a default position that digital moniotrs of all types support a color transfer function similar to analog CRT monitors (gamma=2.2) which makes up the majority of the compute display market." This means DVI output devices should benefit from blending in the sRGB color space just like analog monitors.

Revision History

None

Name

    EXT_fog_coord

Name Strings

    GL_EXT_fog_coord

Status

    Shipping (version 1.6)

Version

    $Date: 1999/06/21 19:57:19 $ $Revision: 1.11 $

Number

    149

Dependencies

    OpenGL 1.1 is required.
    The extension is written against the OpenGL 1.2 Specification.

Overview

    This extension allows specifying an explicit per-vertex fog
    coordinate to be used in fog computations, rather than using a
    fragment depth-based fog equation.

Issues

```
  * Should the specified value be used directly as the fog weighting
    factor, or in place of the z input to the fog equations?

        As the z input; more flexible and meets ISV requests.

  * Do we want vertex array entry points? Interleaved array formats?

        Yes for entry points, no for interleaved formats, following the
        argument for secondary_color.

  * Which scalar types should FogCoord accept? The full range, or just
    the unsigned and float versions? At the moment it follows Index(),
    which takes unsigned byte, signed short, signed int, float, and
    double.

        Since we're now specifying a number which behaves like an
        eye-space distance, rather than a [0,1] quantity, integer types
        are less useful. However, restricting the commands to floating
        point forms only introduces some nonorthogonality.

        Restrict to only float and double, for now.
  * Interpolation of the fog coordinate may be perspective-correct or
    not. Should this be affected by PERSPECTIVE_CORRECTION_HINT,
    FOG_HINT, or another to-be-defined hint?

        PERSPECTIVE_CORRECTION_HINT; this is already defined to affect
        all interpolated parameters. Admittedly this is a loss of
        orthogonality.

  * Should the current fog coordinate be queryable?

        Yes, but it's not returned by feedback.

  * Control the fog coordinate source via an Enable instead of a fog
    parameter?

        No. We might want to add more sources later.

  * Should the fog coordinate be restricted to non-negative values?

        Perhaps. Eye-coordinate distance of fragments will be
        non-negative due to clipping. Specifying explicit negative
        coordinates may result in very large computed f values, although
        they are defined to be clipped after computation.

  * Use existing DEPTH enum instead of FRAGMENT_DEPTH? Change name of
    FRAGMENT_DEPTH_EXT to FOG_FRAGMENT_DEPTH_EXT?

        Use FRAGMENT_DEPTH_EXT; FOG_FRAGMENT_DEPTH_EXT is somewhat
        misleading, since fragment depth itself has no dependence on
        fog.
```

New Procedures and Functions

```
    void FogCoord[fd]EXT(T coord)
    void FogCoord[fd]vEXT(T coord)
    void FogCoordPointerEXT(enum type, sizei stride, void *pointer)
```

New Tokens

```
    Accepted by the <pname> parameter of Fogi and Fogf:

        FOG_COORDINATE_SOURCE_EXT           0x8450

    Accepted by the <param> parameter of Fogi and Fogf:

        FOG_COORDINATE_EXT                  0x8451
        FRAGMENT_DEPTH_EXT                  0x8452

    Accepted by the <pname> parameter of GetBooleanv, GetIntegerv,
    GetFloatv, and GetDoublev:

        CURRENT_FOG_COORDINATE_EXT          0x8453
        FOG_COORDINATE_ARRAY_TYPE_EXT       0x8454
        FOG_COORDINATE_ARRAY_STRIDE_EXT     0x8455
    Accepted by the <pname> parameter of GetPointerv:

        FOG_COORDINATE_ARRAY_POINTER_EXT    0x8456

    Accepted by the <array> parameter of EnableClientState and
    DisableClientState:

        FOG_COORDINATE_ARRAY_EXT            0x8457
```

Additions to Chapter 2 of the OpenGL 1.2 Specification (OpenGL Operation)

```
  These changes describe a new current state type, the fog coordinate,
  and the commands to specify it:

  - (2.6, p. 12) Second paragraph changed to:

        "Each vertex is specified with two, three, or four coordinates.
        In addition, a current normal, current texture coordinates,
        current color, and current fog coordinate may be used in
        processing each vertex."

  - 2.6.3, p. 19) First paragraph changed to

        "The only GL commands that are allowed within any Begin/End
        pairs are the commands for specifying vertex coordinates, vertex
        colors, normal coordinates, texture coordinates, and fog
        coordinates (Vertex, Color, Index, Normal, TexCoord,
        FogCoord)..."

  - (2.7, p. 20) Insert the following paragraph following the third
        paragraph describing current normals:

        "   The current fog coodinate is set using
                void FogCoord[fd]EXT(T coord)
                void FogCoord[fd]vEXT(T coord)."

    The last paragraph is changed to read:

        "The state required to support vertex specification consists of
        four floating-point numbers to store the current texture
        coordinates s, t, r, and q, one floating-point value to store
        the current fog coordinate, four floating-point values to store
        the current RGBA color, and one floating-point value to store
        the current color index. There is no notion of a current vertex,
        so no state is devoted to vertex coordinates. The initial values
        of s, t, and r of the current texture coordinates are zero; the
        initial value of q is one. The initial fog coordinate is zero.
        The initial current normal has coordinates (0,0,1). The initial
        RGBA color is (R,G,B,A) = (1,1,1,1). The initial color index is
        1."

  - (2.8, p. 21) Added fog coordinate command for vertex arrays:

    Change first paragraph to read:

        "The vertex specification commands described in section 2.7
        accept data in almost any format, but their use requires many
        command executions to specify even simple geometry. Vertex data
        may also be placed into arrays that are stored in the client's
        address space. Blocks of data in these arrays may then be used
        to specify multiple geometric primitives through the execution
        of a single GL command. The client may specify up to seven
        arrays: one each to store edge flags, texture coordinates, fog
        coordinates, colors, color indices, normals, and vertices. The
        commands"

    Add to functions listed following first paragraph:

        void FogCoordPointerEXT(enum type, sizei stride, void *pointer)

    Add to table 2.4 (p. 22):

        Command                     Sizes   Types
        -------                     -----   -----
        FogCoordPointerEXT          1       float,double

    Starting with the second paragraph on p. 23, change to add
    FOG_COORDINATE_ARRAY_EXT:

        "An individual array is enabled or disabled by calling one of

            void EnableClientState(enum array)
            void DisableClientState(enum array)

        with array set to EDGE_FLAG_ARRAY, TEXTURE_COORD_ARRAY,
        FOG_COORDINATE_ARRAY_EXT, COLOR_ARRAY, INDEX_ARRAY,
        NORMAL_ARRAY, or VERTEX_ARRAY, for the edge flag, texture
        coordinate, fog coordinate, color, color index, normal, or
        vertex array, respectively.

        The ith element of every enabled array is transferred to the GL
        by calling

            void ArrayElement(int i)

        For each enabled array, it is as though the corresponding
        command from section 2.7 or section 2.6.2 were called with a
        pointer to element i. For the vertex array, the corresponding
        command is Vertex<size><type>v, where <size> is one of [2,3,4],
        and <type> is one of [s,i,f,d], corresponding to array types
        short, int, float, and double respectively. The corresponding
        commands for the edge flag, texture coordinate, fog coordinate,
        color, color, color index, and normal arrays are EdgeFlagv,
        TexCoord<size><type>v, FogCoord<type>v, Color<size><type>v,
        Index<type>v, and Normal<type>v, respectively..."
```

```
   Change pseudocode on p. 27 to disable fog coordinate array for
   canned interleaved array formats. After the lines

           DisableClientState(EDGE_FLAG_ARRAY);
           DisableClientState(INDEX_ARRAY);

       insert the line

           DisableClientState(FOG_COORDINATE_ARRAY_EXT);

   Substitute "seven" for every occurence of "six" in the final
   paragraph on p. 27.

 - (2.12, p. 41) Add fog coordinate to the current rasterpos state.

   Change the first sentence of the first paragraph to read

       "The state required for the current raster position consists of
       three window coordinates x_w, y_w, and z_w, a clip coordinate
       w_c value, an eye coordinate distance, a fog coordinate, a valid
       bit, and associated data consisting of a color and texture
       coordinates."

   Change the last paragraph to read

       "The current raster position requires six single-precision
       floating-point values for its x_w, y_w, and z_w window
       coordinates, its w_c clip coordinate, its eye coordinate
       distance, and its fog coordinate, a single valid bit, a color
       (RGBA color and color index), and texture coordinates for
       associated data. In the initial state, the coordinates and
       texture coordinates are both (0,0,0,1), the fog coordinate is 0,
       the eye coordinate distance is 0, the valid bit is set, the
       associated RGBA color is (1,1,1,1), and the associated color
       index color is 1. In RGBA mode, the associated color index
       always has its initial value; in color index mode, the RGBA
       color always maintains its initial value."

 - (3.10, p. 139) Change the second and third paragraphs to read

       "This factor f may be computed according to one of three
       equations:"

           f = exp(-d*c)       (3.24)
           f = exp(-(d*c)^2)   (3.25)
           f = (e-c)/(e-s)     (3.26)

       If the fog source (as defined below) is FRAGMENT_DEPTH_EXT, then
       c is the eye-coordinate distance from the eye, (0 0 0 1) in eye
       coordinates, to the fragment center. If the fog source is
       FOG_COORDINATE_EXT, then c is the interpolated value of the fog
       coordinate for this fragment. The equation and the fog source,
       along with either d or e and s, is specified with

           void Fog{if}(enum pname, T param);
           void Fog{if}v(enum pname, T params);
```

```
        If <pname> is FOG_MODE, then <param> must be, or <param> must
        point to an integer that is one of the symbolic constants EXP,
        EXP2, or LINEAR, in which case equation 3.24, 3.25, or 3.26,,
        respectively, is selected for the fog calculation (if, when 3.26
        is selected, e = s, results are undefined). If <pname> is
        FOG_COORDINATE_SOURCE_EXT, then <param> is or <params> points to
        an integer that is one of the symbolic constants
        FRAGMENT_DEPTH_EXT or FOG_COORDINATE_EXT. If <pname> is
        FOG_DENSITY, FOG_START, or FOG_END, then <param> is or <params>
        points to a value that is d, s, or e, respectively. If d is
        specified less than zero, the error INVALID_VALUE results."

  - (3.10, p. 140) Change the last paragraph preceding section 3.11
    to read

        "The state required for fog consists of a three valued integer
        to select the fog equation, three floating-point values d, e,
        and s, an RGBA fog color and a fog color index, a two-valued
        integer to select the fog coordinate source, and a single bit to
        indicate whether or not fog is enabled. In the initial state,
        fog is disabled, FOG_COORDINATE_SOURCE_EXT is
        FRAGMENT_DEPTH_EXT, FOG_MODE is EXP, d = 1.0, e = 1.0, and s =
        0.0; C_f = (0,0,0,0) and i_f=0."
```

Additions to Chapter 3 of the OpenGL 1.2.1 Specification (Rasterization)

    None

Additions to Chapter 4 of the OpenGL 1.2.1 Specification (Per-Fragment Operations and the Frame Buffer)

    None

Additions to Chapter 5 of the OpenGL 1.2.1 Specification (Special Functions)

    None

Additions to Chapter 6 of the OpenGL 1.2 Specification (State and State Requests)

    None

Additions to Appendix A of the OpenGL 1.2.1 Specification (Invariance)

    None

Additions to the GLX / WGL / AGL Specifications

    None

GLX Protocol

    Two new GL rendering commands are added. The following commands are
    sent to the server as part of a glXRender request:

```
        FogCoordfvEXT
            2           8               rendering command length
            2           4124            rendering command opcode
            4           FLOAT32         v[0]

        FogCoorddvEXT
            2           12              rendering command length
            2           4125            rendering command opcode
            8           FLOAT64         v[0]
```

Errors

    INVALID_ENUM is generated if FogCoordPointerEXT parameter <type> is
    not FLOAT or DOUBLE.

    INVALID_VALUE is generated if FogCoordPointerEXT parameter <stride>
    is negative.

New State

(table 6.5, p. 195)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
CURRENT_FOG_COORDINATE_EXT	R	GetIntegerv, GetFloatv	0	Current fog coordinate	2.7	current

(table 6.6, p. 197)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
FOG_COORDINATE_ARRAY_EXT	B	IsEnabled	False	Fog coord array enable	2.8	vertex-array
FOG_COORDINATE_ARRAY_TYPE_EXT	Z8	GetIntegerv	FLOAT	Type of fog coordinate	2.8	vertex-array
FOG_COORDINATE_ARRAY_STRIDE_EXT	Z+	GetIntegerv	0	Stride between fog coords	2.8	vertex-array
FOG_COORDINATE_ARRAY_POINTER_EXT	Y	GetPointerv	0	Pointer to the fog coord array	2.8	vertex-array

(table 6.8, p. 198)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
FOG_COORDINATE_SOURCE_EXT	Z2	GetIntegerv, GetFloatv	FRAGMENT_DEPTH_EXT	Source of fog coordinate for fog calculation	3.10	fog

Revision History

* Revision 1.6 - Functionality complete

* Revision 1.7-1.9 - Fix typos and add fields to bring up to date with
  the new extension template. No functionality changes.

Name

    EXT_geometry_shader4

Name String

    GL_EXT_geometry_shader4

Contact

    Pat Brown, NVIDIA (pbrown 'at' nvidia.com)
    Barthold Lichtenbelt, NVIDIA (blichtenbelt 'at' nvidia.com)

Status

    Multi-vendor extension

    Shipping for GeForce 8 Series (November 2006)

Version

    Last Modified Date: 05/22/2007
    NVIDIA Revision: 17

Number

    324

Dependencies

    OpenGL 1.1 is required.

    This extension is written against the OpenGL 2.0 specification.

    EXT_framebuffer_object interacts with this extension.

    EXT_framebuffer_blit interacts with this extension.

    EXT_texture_array interacts with this extension.

    ARB_texture_rectangle trivially affects the definition of this
    extension.

    EXT_texture_buffer_object trivially affects the definition of this
    extension.

    NV_primitive_restart trivially affects the definition of this
    extension.

    This extension interacts with EXT_tranform_feedback.

Overview

    EXT_geometry_shader4 defines a new shader type available to be run on the
    GPU, called a geometry shader. Geometry shaders are run after vertices are
    transformed, but prior to color clamping, flat shading and clipping.

A geometry shader begins with a single primitive (point, line, triangle). It can read the attributes of any of the vertices in the primitive and use them to generate new primitives. A geometry shader has a fixed output primitive type (point, line strip, or triangle strip) and emits vertices to define a new primitive. A geometry shader can emit multiple disconnected primitives. The primitives emitted by the geometry shader are clipped and then processed like an equivalent OpenGL primitive specified by the application.

Furthermore, EXT_geometry_shader4 provides four additional primitive types: lines with adjacency, line strips with adjacency, separate triangles with adjacency, and triangle strips with adjacency. Some of the vertices specified in these new primitive types are not part of the ordinary primitives, instead they represent neighboring vertices that are adjacent to the two line segment end points (lines/strips) or the three triangle edges (triangles/tstrips). These vertices can be accessed by geometry shaders and used to match up the vertices emitted by the geometry shader with those of neighboring primitives.

Since geometry shaders expect a specific input primitive type, an error will occur if the application presents primitives of a different type. For example, if a geometry shader expects points, an error will occur at Begin() time, if a primitive mode of TRIANGLES is specified.

New Procedures and Functions

```
void ProgramParameteriEXT(uint program, enum pname, int value);
void FramebufferTextureEXT(enum target, enum attachment,
                           uint texture, int level);
void FramebufferTextureLayerEXT(enum target, enum attachment,
                                uint texture, int level, int layer);
void FramebufferTextureFaceEXT(enum target, enum attachment,
                               uint texture, int level, enum face);
```

New Tokens

Accepted by the <type> parameter of CreateShader and returned by the <params> parameter of GetShaderiv:

```
    GEOMETRY_SHADER_EXT                             0x8DD9
```

Accepted by the <pname> parameter of ProgramParameteriEXT and GetProgramiv:

```
    GEOMETRY_VERTICES_OUT_EXT                       0x8DDA
    GEOMETRY_INPUT_TYPE_EXT                         0x8DDB
    GEOMETRY_OUTPUT_TYPE_EXT                        0x8DDC
Accepted by the <pname> parameter of GetBooleanv, GetIntegerv,
GetFloatv, and GetDoublev:

    MAX_GEOMETRY_TEXTURE_IMAGE_UNITS_EXT             0x8C29
    MAX_GEOMETRY_VARYING_COMPONENTS_EXT              0x8DDD
    MAX_VERTEX_VARYING_COMPONENTS_EXT                0x8DDE
    MAX_VARYING_COMPONENTS_EXT                       0x8B4B
    MAX_GEOMETRY_UNIFORM_COMPONENTS_EXT              0x8DDF
    MAX_GEOMETRY_OUTPUT_VERTICES_EXT                 0x8DE0
    MAX_GEOMETRY_TOTAL_OUTPUT_COMPONENTS_EXT         0x8DE1

Accepted by the <mode> parameter of Begin, DrawArrays,
MultiDrawArrays, DrawElements, MultiDrawElements, and
DrawRangeElements:

    LINES_ADJACENCY_EXT                              0xA
    LINE_STRIP_ADJACENCY_EXT                         0xB
    TRIANGLES_ADJACENCY_EXT                          0xC
    TRIANGLE_STRIP_ADJACENCY_EXT                     0xD

Returned by CheckFramebufferStatusEXT:

    FRAMEBUFFER_INCOMPLETE_LAYER_TARGETS_EXT         0x8DA8
    FRAMEBUFFER_INCOMPLETE_LAYER_COUNT_EXT           0x8DA9

Accepted by the <pname> parameter of GetFramebufferAttachment-
ParameterivEXT:

    FRAMEBUFFER_ATTACHMENT_LAYERED_EXT               0x8DA7
    FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER_EXT         0x8CD4

Accepted by the <cap> parameter of Enable, Disable, and IsEnabled,
and by the <pname> parameter of GetIntegerv, GetFloatv, GetDoublev,
and GetBooleanv:

    PROGRAM_POINT_SIZE_EXT                           0x8642

(Note: FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER_EXT is simply an alias for the
FRAMEBUFFER_ATTACHMENT_TEXTURE_3D_ZOFFSET_EXT token provided in
EXT_framebuffer_object.  This extension generalizes the notion of
"<zoffset>" to include layers of an array texture.)

(Note:  PROGRAM_POINT_SIZE_EXT is simply an alias for the
VERTEX_PROGRAM_POINT_SIZE token provided in OpenGL 2.0, which is itself an
alias for VERTEX_PROGRAM_POINT_SIZE_ARB provided by
ARB_vertex_program. Program-computed point sizes can be enabled if
geometry shaders are enabled.)
```

Additions to Chapter 2 of the OpenGL 2.0 Specification (OpenGL Operation)

Modify Section 2.6.1 (Begin and End Objects), p. 13

(Add to end of section, p. 18)

(add figure)

```
    1 - - - 2----->3 - - - 4     1 - - - 2--->3--->4--->5 - - - 6

    5 - - - 6----->7 - - - 8

           (a)                              (b)
```

Figure 2.X1 (a) Lines with adjacency, (b) Line strip with adjacency. The vertices connected with solid lines belong to the main primitives; the vertices connected by dashed lines are the adjacent vertices that may be used in a geometry shader.

Lines with Adjacency

Lines with adjacency are independent line segments where each endpoint has a corresponding "adjacent" vertex that can be accessed by a geometry shader (Section 2.16). If a geometry shader is not active, the "adjacent" vertices are ignored.

A line segment is drawn from the 4i + 2nd vertex to the 4i + 3rd vertex for each i = 0, 1, ... , n-1, where there are 4n+k vertices between the Begin and End. k is either 0, 1, 2, or 3; if k is not zero, the final k vertices are ignored. For line segment i, the 4i + 1st and 4i + 4th vertices are considered adjacent to the 4i + 2nd and 4i + 3rd vertices, respectively. See Figure 2.X1.

Lines with adjacency are generated by calling Begin with the argument value LINES_ADJACENCY_EXT.

Line Strips with Adjacency

Line strips with adjacency are similar to line strips, except that each line segment has a pair of adjacent vertices that can be accessed by a geometry shader (Section 2.15). If a geometry shader is not active, the "adjacent" vertices are ignored.

A line segment is drawn from the i + 2nd vertex to the i + 3rd vertex for each i = 0, 1, ..., n-1, where there are n+3 vertices between the Begin and End. If there are fewer than four vertices between a Begin and End, all vertices are ignored. For line segment i, the i + 1st and i + 4th vertex are considered adjacent to the i + 2nd and i + 3rd vertices, respectively. See Figure 2.X1.

Line strips with adjacency are generated by calling Begin with the argument value LINE_STRIP_ADJACENCY_EXT.

```
(add figure)

               2 - - - 3 - - - 4     8 - - - 9 - - - 10
                       ^\                    ^\
                 \     | \     |       \     | \     |
                       |  \                  |  \
                   \   |   \   |         \   |   \   |
                       |    \                |    \
                     \ |     \ |           \ |     \ |
                       |      v              |      v
                       1<------5             7<------11

                         \     |               \     |

                           \   |                 \   |

                             \ |                   \ |

                               6                     12
```

Figure 2.X2 Triangles with adjacency. The vertices connected with solid lines belong to the main primitive; the vertices connected by dashed lines are the adjacent vertices that may be used in a geometry shader.

Triangles with Adjacency

Triangles with adjacency are similar to separate triangles, except that each triangle edge has an adjacent vertex that can be accessed by a geometry shader (Section 2.15). If a geometry shader is not active, the "adjacent" vertices are ignored.

The 6i + 1st, 6i + 3rd, and 6i + 5th vertices (in that order) determine a triangle for each i = 0, 1, ..., n-1, where there are 6n+k vertices between the Begin and End. k is either 0, 1, 2, 3, 4, or 5; if k is non-zero, the final k vertices are ignored. For triangle i, the i + 2nd, i + 4th, and i + 6th vertices are considered adjacent to edges from the i + 1st to the i + 3rd, from the i + 3rd to the i + 5th, and from the i + 5th to the i + 1st vertices, respectively. See Figure 2.X2.

Triangles with adjacency are generated by calling Begin with the argument value TRIANGLES_ADJACENCY_EXT.

```
  (add figure)
                                6                     6

                                | \                   | \

                                |   \                 |   \

                                |     \               |     \

    2 - - - 3- - - >6   2 - - - 3------>7     2 - - - 3------>7- - - 10
            ^\                  ^^      |             ^^      ^^      |
      \     | \     |     \     | \     | \     \     | \     | \
            |  \                |  \    |             |  \    |  \    |
        \   |   \   |       \   |   \   |   \     \   |   \   |   \
            |    \              |    \  |             |    \  |    \  |
          \ |     \ |         \ |     \ |     \     \ |     \ |     \
            |      v            |      vv             |      vv      v|
            1<------5           1<------5 - - - 8     1<------5<------9

              \     |             \     |               \     | \     |

                \   |               \   |                 \   |   \   |

                  \ |                 \ |                   \ |     \ |

                    4                   4                     4       8


                                 6       10

                                 | \     | \

                                 |   \   |   \

                                 |     \ |     \
                         2 - - - 3------>7------>11
                                 ^^      ^^      |
                           \     | \     | \     | \
                                 |  \    |  \    |
                             \   |   \   |   \   |   \
                                 |    \  |    \  |
                               \ |     \ |     \ |     \
                                 |      vv      vv
                                 1<------5<------9 - - - 12

                                   \     | \     |

                                     \   |   \   |

                                       \ |     \ |

                                         4       8
```

Figure 2.X3 Triangle strips with adjacency. The vertices connected with solid lines belong to the main primitives; the vertices connected by dashed lines are the adjacent vertices that may be used in a geometry shader.

Triangle Strips with Adjacency

Triangle strips with adjacency are similar to triangle strips, except that each line triangle edge has an adjacent vertex that can be accessed by a geometry shader (Section 2.15). If a geometry shader is not active, the "adjacent" vertices are ignored.

In triangle strips with adjacency, n triangles are drawn using 2 * (n+2) + k vertices between the Begin and End. k is either 0 or 1; if k is 1, the final vertex is ignored. If fewer than 6 vertices are specified between the Begin and End, the entire primitive is ignored. Table 2.X1 describes the vertices and order used to draw each triangle, and which vertices are considered adjacent to each edge of the triangle. See Figure 2.X3.

(add table)

	primitive vertices			adjacent vertices		
primitive	1st	2nd	3rd	1/2	2/3	3/1
only (i==0, n==1)	1	3	5	2	6	4
first (i==0)	1	3	5	2	7	4
middle (i odd)	2i+3	2i+1	2i+5	2i-1	2i+4	2i+7
middle (i even)	2i+1	2i+3	2i+5	2i-1	2i+7	2i+4
last (i==n-1, i odd)	2i+3	2i+1	2i+5	2i-1	2i+4	2i+6
last (i==n-1, i even)	2i+1	2i+3	2i+5	2i-1	2i+6	2i+4

Table 2.X1: Triangles generated by triangle strips with adjacency. Each triangle is drawn using the vertices in the "1st", "2nd", and "3rd" columns under "primitive vertices", in that order. The vertices in the "1/2", "2/3", and "3/1" columns under "adjacent vertices" are considered adjacent to the edges from the first to the second, from the second to the third, and from the third to the first vertex of the triangle, respectively. The six rows correspond to the six cases: the first and only triangle (i=0, n=1), the first triangle of several (i=0, n>0), "odd" middle triangles (i=1,3,5...), "even" middle triangles (i=2,4,6,...), and special cases for the last triangle inside the Begin/End, when i is either even or odd. For the purposes of this table, the first vertex specified after Begin is numbered "1" and the first triangle is numbered "0".

Triangle strips with adjacency are generated by calling Begin with the argument value TRIANGLE_STRIP_ADJACENCY_EXT.

Modify Section 2.14.1, Lighting (p. 59)

(modify fourth paragraph, p. 63) Additionally, vertex and geometry shaders can operate in two-sided color mode, which is enabled and disabled by calling Enable or Disable with the symbolic value VERTEX_PROGRAM_TWO_SIDE. When a vertex or geometry shader is active, the shaders can write front and back color values to the gl_FrontColor, gl_BackColor, gl_FrontSecondaryColor and gl_BackSecondaryColor outputs. When a vertex or geometry shader is active and two-sided color mode is enabled, the GL chooses between front and back colors, as described below. If two-sided color mode is disabled, the front color output is always selected.

Modify Section 2.15.2 Program Objects, p. 73

Change the first paragraph on p. 74 as follows:

Program objects are empty when they are created. Default values for program object parameters are discussed in section 2.15.5, Required State. A non-zero name that can be used to reference the program object is returned.

Change the language below the LinkProgram command on p. 74 as follows:

... Linking can fail for a variety of reasons as specified in the OpenGL Shading Language Specification. Linking will also fail if one or more of the shader objects, attached to <program> are not compiled successfully, or if more active uniform or active sampler variables are used in <program> than allowed (see sections 2.15.3 and 2.16.3). Linking will also fail if the program object contains objects to form a geometry shader (see section 2.16), but no objects to form a vertex shader or if the program object contains objects to form a geometry shader, and the value of GEOMETRY_VERTICES_OUT_EXT is zero. If LinkProgram failed, ..

Add the following paragraphs above the description of DeleteProgram, p. 75:

To set a program object parameter, call

```
    void ProgramParameteriEXT(uint program, enum pname, int value)
```

<param> identifies which parameter to set for <program>. <value> holds the value being set. Legal values for <param> and <value> are discussed in section 2.16.

Modify Section 2.15.3, Shader Variables, p. 75

Modify the first paragraph of section 'Varying Variables' p. 83 as follows:

A vertex shader may define one or more varying variables (see the OpenGL Shading Language specification). Varying variables are outputs of a vertex shader. They are either used as the mechanism to communicate values to a geometry shader, if one is active, or to communicate values to the fragment shader. The OpenGL Shading Language specification also defines a set of built-in varying variables that vertex shaders can write to (see section 7.6 of the OpenGL Shading Language Specification). These variables can also be used to communicate values to a geometry shader, if one is active, or to communicate values to the fragment shader and to the fixed-function processing that occurs after vertex shading.

If a geometry shader is not active, the values of all varying variables, including built-in variables, are expected to be interpolated across the primitive being rendered, unless flat shaded. The number of interpolators available for processing varying variables is given by the implementation-dependent constant MAX_VARYING_COMPONENTS_EXT. This value represents the number of individual components that can be interpolated; varying variables declared as vectors, matrices, and arrays will all consume multiple interpolators. When a program is linked, all components of any varying variable written by a vertex shader, or read by a fragment shader, will count against this limit. The transformed vertex position (gl_Position) does not count against this limit. A program whose vertex and/or fragment shaders access more than MAX_VARYING_COMPONENTS_EXT components worth of varying variables may fail to link, unless device-dependent optimizations are able to make the program fit within available hardware resources.

Note that the two values MAX_VARYING_FLOATS and MAX_VARYING_COMPONENTS_EXT are aliases of each other. The use of MAX_VARYING_FLOATS however is discouraged; varying variables can be declared as integers as well.

If a geometry shader is active, the values of varying variables are collected by the primitive assembly stage and passed on to the geometry shader once enough data for one primitive has been collected (see also section 2.16). The OpenGL Shading Language specification also defines a set of built-in varying and built-in special variables that vertex shaders can write to (see sections 7.1 and 7.6 of the OpenGL Shading Language Specification). These variables are also collected and passed on to the geometry shader once enough data has been collected. The number of components of varying and special variables that can be collected per vertex by the primitive assembly stage is given by the implementation dependent constant MAX_VERTEX_VARYING_COMPONENTS_EXT. This value represents the number of individual components that can be collected; varying variables declared as vectors, matrices, and arrays will all consume multiple components. When a program is linked, all components of any varying variable written by a vertex shader, or read by a geometry shader, will count against this limit. A program whose vertex and/or geometry shaders access more than MAX_VERTEX_VARYING_COMPONENTS_EXT components worth of varying variables may fail to link, unless device-dependent optimizations are able to make the program fit within available hardware resources.

Modify Section 2.15.4 Shader Execution, p. 84

Change the following sentence:

"The following operations are applied to vertex values that are the result of executing the vertex shader:"

As follows:

If no geometry shader (see section 2.16) is present in the program object, the following operations are applied to vertex values that are the result of executing the vertex shader:

[bulleted list of operations]

On page 85, below the list of bullets, add the following:

If a geometry shader is present in the program object, geometry shading (section 2.16) is applied to vertex values that are the result of executing the vertex shader.

Modify the first paragraph of the section 'Texture Access', p. 85, as follows:

Vertex shaders have the ability to do a lookup into a texture map, if supported by the GL implementation. The maximum number of texture image units available to a vertex shader is MAX_VERTEX_TEXTURE_IMAGE_UNITS; a maximum number of zero indicates that the GL implementation does not support texture accesses in vertex shaders. The vertex shader, geometry shader, if exists, and fragment processing combined cannot use more than MAX_COMBINED_TEXTURE_IMAGE_UNITS texture image units. If the vertex shader, geometry shader and the fragment processing stage access the same texture image unit, then that counts as using three texture image units against the MAX_COMBINED_TEXTURE_IMAGE_UNITS limit.

Modify Section 2.15.5, Required State, p. 88

Add the following bullets to the state required per program object:

* One integer to store the value of GEOMETRY_VERTICES_OUT_EXT, initially zero.

* One integer to store the value of GEOMETRY_INPUT_TYPE_EXT, initially set to TRIANGLES.

* One integer to store the value of GEOMETRY_OUTPUT_TYPE_EXT, initially set to TRIANGLE_STRIP.

Insert New Section 2.16, Geometry Shaders after p. 89

After vertices are processed, they are arranged into primitives, as described in section 2.6.1 (Begin/End Objects). This section described a new pipeline stage that processes those primitives. A geometry shader defines the operations that are performed in this new pipeline stage. A geometry shader is an array of strings containing source code. The source code language used is described in the OpenGL Shading Language specification. A geometry shader operates on a single primitive at a time and emits one or more output primitives, all of the same type, which are then processed like an equivalent OpenGL primitive specified by the application. The original primitive is discarded after the geometry shader completes. The inputs available to a geometry shader are the transformed attributes of all the vertices that belong to the primitive. Additional "adjacency" primitives are available which also make the transformed attributes of neighboring vertices available to the shader. The results of the shader are a new set of transformed vertices, arranged into primitives by the shader.

This new geometry shader pipeline stage is inserted after primitive assembly, right before color clamping (section 2.14.6), flat shading (section 2.14.7) and clipping (sections 2.12 and 2.14.8).

A geometry shader only applies when the GL is in RGB mode. Its operation in color index mode is undefined.

Geometry shaders are created as described in section 2.15.1 using a type parameter of GEOMETRY_SHADER_EXT. They are attached to and used in program objects as described in section 2.15.2. When the program object currently in use includes a geometry shader, its geometry shader is considered active, and is used to process primitives. If the program object has no geometry shader, or no program object is in use, this new primitive processing pipeline stage is bypassed.

A program object that includes a geometry shader must also include a vertex shader; otherwise a link error will occur.

Section 2.16.1, Geometry shader Input Primitives

A geometry shader can operate on one of five input primitive types. Depending on the input primitive type, one to six input vertices are available when the shader is executed. Each input primitive type supports a subset of the primitives provided by the GL. If a geometry shader is active, Begin, or any function that implicitly calls Begin, will produce an INVALID_OPERATION error if the <mode> parameter is incompatible with the input primitive type of the currently active program object, as discussed below.

The input primitive type is a parameter of the program object, and must be set before linking by calling ProgramParameteriEXT with <pname> set to GEOMETRY_INPUT_TYPE_EXT and <value> set to one of POINTS, LINES, LINES_ADJACENCY_EXT, TRIANGLES or TRIANGLES_ADJACENCY_EXT. This setting will not be in effect until the next time LinkProgram has been called successfully. Note that queries of GEOMETRY_INPUT_TYPE_EXT will return the last value set. This is not necessarily the value used to generate the executable code in the program object. After a program object has been created it will have a default value for GEOMETRY_INPUT_TYPE_EXT, as discussed in section 2.15.5, Required State.

Note that a geometry shader that accesses more input vertices than are available for a given input primitive type can be successfully compiled, because the input primitive type is not part of the shader object. However, a program object, containing a shader object that access more input vertices than are available for the input primitive type of the program object, will not link.

The supported input primitive types are:

Points (POINTS)

Geometry shaders that operate on points are valid only for the POINTS primitive type. There is only a single vertex available for each geometry shader invocation.

Lines (LINES)

Geometry shaders that operate on line segments are valid only for the LINES, LINE_STRIP, and LINE_LOOP primitive types. There are two vertices available for each geometry shader invocation. The first vertex refers to the vertex at the beginning of the line segment and the second vertex refers to the vertex at the end of the line segment. See also section 2.16.4.

Lines with Adjacency (LINES_ADJACENCY_EXT)

Geometry shaders that operate on line segments with adjacent vertices are valid only for the LINES_ADJACENCY_EXT and LINE_STRIP_ADJACENCY_EXT primitive types. There are four vertices available for each program invocation. The second vertex refers to attributes of the vertex at the beginning of the line segment and the third vertex refers to the vertex at the end of the line segment. The first and fourth vertices refer to the vertices adjacent to the beginning and end of the line segment, respectively.

Triangles (TRIANGLES)

Geometry shaders that operate on triangles are valid for the TRIANGLES, TRIANGLE_STRIP and TRIANGLE_FAN primitive types.

There are three vertices available for each program invocation. The first, second and third vertices refer to attributes of the first, second and third vertex of the triangle, respectively.

Triangles with Adjacency (TRIANGLES_ADJACENCY_EXT)

Geometry shaders that operate on triangles with adjacent vertices are valid for the TRIANGLES_ADJACENCY_EXT and TRIANGLE_STRIP_ADJACENCY_EXT primitive types. There are six vertices available for each program invocation. The first, third and fifth vertices refer to attributes of the first, second and third vertex of the triangle, respectively. The second, fourth and sixth vertices refer to attributes of the vertices adjacent to the edges from the first to the second vertex, from the second to the third vertex, and from the third to the first vertex, respectively.

Section 2.16.2, Geometry Shader Output Primitives

A geometry shader can generate primitives of one of three types. The supported output primitive types are points (POINTS), line strips (LINE_STRIP), and triangle strips (TRIANGLE_STRIP). The vertices output by the geometry shader are decomposed into points, lines, or triangles based on the output primitive type in the manner described in section 2.6.1. The resulting primitives are then further processed as shown in figure 2.16.xxx. If the number of vertices emitted by the geometry shader is not sufficient to produce a single primitive, nothing is drawn.

The output primitive type is a parameter of the program object, and can be set by calling ProgramParameteriEXT with <pname> set to GEOMETRY_OUTPUT_TYPE_EXT and <value> set to one of POINTS, LINE_STRIP or TRIANGLE_STRIP. This setting will not be in effect until the next time LinkProgram has been called successfully. Note that queries of GEOMETRY_OUTPUT_TYPE_EXT will return the last value set; which is not necessarily the value used to generate the executable code in the program object. After a program object has been created it will have a default value for GEOMETRY_OUTPUT_TYPE_EXT, as discussed in section 2.15.5, Required State. .

Section 2.16.3 Geometry Shader Variables

Geometry shaders can access uniforms belonging to the current program object. The amount of storage available for geometry shader uniform variables is specified by the implementation dependent constant MAX_GEOMETRY_UNIFORM_COMPONENTS_EXT. This value represents the number of individual floating-point, integer, or Boolean values that can be held in uniform variable storage for a geometry shader. A link error will be generated if an attempt is made to utilize more than the space available for geometry shader uniform variables. Uniforms are manipulated as described in section 2.15.3. Geometry shaders also have access to samplers, to perform texturing operations, as described in sections 2.15.3 and 3.8.

Geometry shaders can access the transformed attributes of all vertices for its input primitive type through input varying variables. A vertex shader, writing to output varying variables, generates the values of these input varying variables. This includes values for built-in as well as user-defined varying variables. Values for any varying variables that are not written by a vertex shader are undefined. Additionally, a geometry shader has access to a built-in variable that holds the ID of the current primitive. This ID is generated by the primitive assembly stage that sits in between the vertex and geometry shader.

Additionally, geometry shaders can write to one, or more, varying variables for each primitive it outputs. These values are optionally flat shaded (using the OpenGL Shading Language varying qualifier "flat") and clipped, then the clipped values interpolated across the primitive (if not flat shaded). The results of these interpolations are available to a fragment shader, if one is active. Furthermore, geometry shaders can write to a set of built- in varying variables, defined in the OpenGL Shading Language, that correspond to the values required for the fixed-function processing that occurs after geometry processing.

Section 2.16.4, Geometry Shader Execution Environment

If a successfully linked program object that contains a geometry shader is made current by calling UseProgram, the executable version of the geometry shader is used to process primitives resulting from the primitive assembly stage.

The following operations are applied to the primitives that are the result of executing a geometry shader:

* color clamping or masking (section 2.14.6),
* flat shading (section 2.14.7),
* clipping, including client-defined clip planes (section 2.12),
* front face determination (section 2.14.1),
* color and associated data clipping (section 2.14.8),
* perspective division on clip coordinates (section 2.11),
* final color processing (section 2.14.9), and
* viewport transformation, including depth-range scaling (section 2.11.1).

There are several special considerations for geometry shader execution described in the following sections.

Texture Access

Geometry shaders have the ability to do a lookup into a texture map, if supported by the GL implementation. The maximum number of texture image units available to a geometry shader is MAX_GEOMETRY_TEXTURE_IMAGE_UNITS_EXT; a maximum number of zero indicates that the GL implementation does not support texture accesses in geometry shaders.

The vertex shader, geometry shader and fragment processing combined cannot use more than MAX_COMBINED_TEXTURE_IMAGE_UNITS texture image units. If the vertex shader, geometry shader and the fragment processing stage access the same texture image unit, then that counts as using three texture image units against the MAX_COMBINED_TEXTURE_IMAGE_UNITS limit.

When a texture lookup is performed in a geometry shader, the filtered texture value tau is computed in the manner described in sections 3.8.8 and 3.8.9, and converted to a texture source color Cs according to table 3.21 (section 3.8.13). A four component vector (Rs,Gs,Bs,As) is returned to the geometry shader. In a geometry shader it is not possible to perform automatic level-of- detail calculations using partial derivatives of the texture coordinates with respect to window coordinates as described in section 3.8.8. Hence, there is no automatic selection of an image array level. Minification or magnification of a texture map is controlled by a level-of-detail value optionally passed as an argument in the texture lookup functions. If the texture lookup function supplies an explicit level-of-detail value lambda, then the pre-bias level-of-detail value LAMBDAbase(x, y) = lambda (replacing equation 3.18). If the texture lookup function does not supply an explicit level-of-detail value, then LAMBDAbase(x, y) = 0. The scale factor Rho(x, y) and its approximation function f(x, y) (see equation 3.21) are ignored.

Texture lookups involving textures with depth component data can either return the depth data directly or return the results of a comparison with the R value (see section 3.8.14) used to perform the lookup. The comparison operation is requested in the shader by using any of the shadow sampler and in the texture using the TEXTURE COMPARE MODE parameter. These requests must be consistent; the results of a texture lookup are undefined if:

* the sampler used in a texture lookup function is not one of the shadow sampler types, and the texture object's internal format is DEPTH COMPONENT, and the TEXTURE COMPARE MODE is not NONE;

* the sampler used in a texture lookup function is one of the shadow sampler types, and the texture object's internal format is DEPTH COMPONENT, and the TEXTURE COMPARE MODE is NONE; or

* the sampler used in a texture lookup function is one of the shadow sampler types, and the texture object's internal format is not DEPTH COMPONENT.

If a geometry shader uses a sampler where the associated texture object is not complete as defined in section 3.8.10, the texture image unit will return (R,G,B,A) = (0, 0, 0, 1).

Geometry Shader Inputs

The OpenGL Shading Language specification describes the set of built-in variables that are available as inputs to the geometry shader. This set receives the values from the equivalent built-in output variables written by the vertex shader. These built-in variables are arrays; each element in the array holds the value for a specific vertex of the input primitive. The length of each array depends on the value of the input primitive type, as determined by the program object value GEOMETRY_INPUT_TYPE_EXT, and is set by the GL during link. Each built-in variable is a one-dimensional array, except for the built-in texture coordinate variable, which is a two- dimensional array. The vertex shader built-in output gl_TexCoord[] is a one-dimensional array. Therefore, the geometry shader equivalent input variable gl_TexCoordIn[][] becomes a two-dimensional array. See the OpenGL Shading Language Specification, sections 4.3.6 and 7.6 for more information.

The built-in varying variables gl_FrontColorIn[], gl_BackColorIn[], gl_FrontSecondaryColorIn[] and gl_BackSecondaryColorIn[] hold the per-vertex front and back colors of the primary and secondary colors, as written by the vertex shader to its equivalent built-in output variables.

The built-in varying variable gl_TexCoordIn[][] holds the per- vertex values of the array of texture coordinates, as written by the vertex shader to its built-in output array gl_TexCoord[].

The built-in varying variable gl_FogFragCoordIn[] holds the per- vertex fog coordinate, as written by the vertex shader to its built- in output variable gl_FogFragCoord.

The built-in varying variable gl_PositionIn[] holds the per-vertex position, as written by the vertex shader to its output variable gl_Position. Note that writing to gl_Position from either the vertex or fragment shader is optional. See also section 7.1 "Vertex and Geometry Shader Special Variables" of the OpenGL Shading Language specification.

The built-in varying variable gl_ClipVertexIn[] holds the per-vertex position in clip coordinates, as written by the vertex shader to its output variable gl_ClipVertex.

The built-in varying variable gl_PointSizeIn[] holds the per-vertex point size written by the vertex shader to its built-in output varying variable gl_PointSize. If the vertex shader does not write gl_PointSize, the value of gl_PointSizeIn[] is undefined, regardless of the value of the enable VERTEX_PROGRAM_POINT_SIZE.

The built-in special variable gl_PrimitiveIDIn is not an array and has no vertex shader equivalent. It is filled with the number of primitives processed since the last time Begin was called (directly or indirectly via vertex array functions). The first primitive generated after a Begin is numbered zero, and the primitive ID counter is incremented after every individual point, line, or triangle primitive is processed. For triangles drawn in point or line mode, the primitive ID counter is incremented only once, even though multiple points or lines may be drawn. Restarting a primitive topology using the primitive restart index has no effect on the primitive ID counter.

Similarly to the built-in varying variables, user-defined input varying variables need to be declared as arrays. Declaring a size is optional. If no size is specified, it will be inferred by the linker from the input primitive type. If a size is specified, it has to be of the size matching the number of vertices of the input primitive type, otherwise a link error will occur. The built-in variable gl_VerticesIn, if so desired, can be used to size the array correctly for each input primitive type. User-defined varying variables can be declared as arrays in the vertex shader. This means that those, on input to the geometry shader, must be declared as two-dimensional arrays. See sections 4.3.6 and 7.6 of the OpenGL Shading Language Specification for more information.

Using any of the built-in or user-defined input varying variables can count against the limit MAX_VERTEX_VARYING_COMPONENTS_EXT as discussed in section 2.15.3.

Geometry Shader outputs

A geometry shader is limited in the number of vertices it may emit per invocation. The maximum number of vertices a geometry shader can possibly emit needs to be set as a parameter of the program object that contains the geometry shader. To do so, call ProgramParameteriEXT with <pname> set to GEOMETRY_VERTICES_OUT_EXT and <value> set to the maximum number of vertices the geometry shader will emit in one invocation. This setting will not be guaranteed to be in effect until the next time LinkProgram has been called successfully. If a geometry shader, in one invocation, emits more vertices than the value GEOMETRY_VERTICES_OUT_EXT, these emits may have no effect.

There are two implementation-dependent limits on the value of GEOMETRY_VERTICES_OUT_EXT. First, the error INVALID_VALUE will be generated by ProgramParameteriEXT if the number of vertices specified exceeds the value of MAX_GEOMETRY_OUTPUT_VERTICES_EXT. Second, the product of the total number of vertices and the sum of all components of all active varying variables may not exceed the value of MAX_GEOMETRY_TOTAL_OUTPUT_COMPONENTS_EXT. LinkProgram will fail if it determines that the total component limit would be violated.

A geometry shader can write to built-in as well as user-defined varying variables. These values are expected to be interpolated across the primitive it outputs, unless they are specified to be flat shaded. In order to seamlessly be able to insert or remove a geometry shader from a program object, the rules, names and types of the output built-in varying variables and user-defined varying variables are the same as for the vertex shader. Refer to section 2.15.3 and the OpenGL Shading Language specification sections 4.3.6, 7.1 and 7.6 for more detail.

The built-in output variables gl_FrontColor, gl_BackColor, gl_FrontSecondaryColor, and gl_BackSecondaryColor hold the front and back colors for the primary and secondary colors for the current vertex.

The built-in output variable gl_TexCoord[] is an array and holds the set of texture coordinates for the current vertex.

The built-in output variable gl_FogFragCoord is used as the "c" value, as described in section 3.10 "Fog" of the OpenGL 2.0 specification.

The built-in special variable gl_Position is intended to hold the homogeneous vertex position. Writing gl_Position is optional.

The built-in special variable gl_ClipVertex holds the vertex coordinate used in the clipping stage, as described in section 2.12 "Clipping" of the OpenGL 2.0 specification.

The built-in special variable gl_PointSize, if written, holds the size of the point to be rasterized, measured in pixels.

Additionally, a geometry shader can write to the built-in special variables gl_PrimitiveID and gl_Layer, whereas a vertex shader cannot. The built-in gl_PrimitiveID provides a single integer that serves as a primitive identifier. This written primitive ID is available to fragment shaders. If a fragment shader using primitive IDs is active and a geometry shader is also active, the geometry shader must write to gl_PrimitiveID or the primitive ID number is undefined. The built-in variable gl_Layer is used in layered rendering, and discussed in the next section.

The number of components available for varying variables is given by the implementation-dependent constant MAX_GEOMETRY_VARYING_COMPONENTS_EXT. This value represents the number of individual components of a varying variable; varying variables declared as vectors, matrices, and arrays will all consume multiple components. When a program is linked, all components of any varying variable written by a geometry shader, or read by a fragment shader, will count against this limit. The transformed vertex position (gl_Position) does not count against this limit. A program whose geometry and/or fragment shaders access more than MAX_GEOMETRY_VARYING_COMPONENTS_EXT worth of varying variable components may fail to link, unless device-dependent optimizations are able to make the program fit within available hardware resources.

Layered rendering

Geometry shaders can be used to render to one of several different layers of cube map textures, three-dimensional textures, plus one- dimensional and two-dimensional texture arrays. This functionality allows an application to bind an entire "complex" texture to a framebuffer object, and render primitives to arbitrary layers computed at run time. For example, this mechanism can be used to project and render a scene onto all six faces of a cubemap texture in one pass. The layer to render to is specified by writing to the built-in output variable gl_layer. Layered rendering requires the use of framebuffer objects. Refer to the section 'Dependencies on EXT_framebuffer_object' for details.

Additions to Chapter 3 of the OpenGL 2.0 Specification (Rasterization)

Modify Section 3.3, Points (p. 95)

(replace all Section 3.3 text on p. 95)

A point is drawn by generating a set of fragments in the shape of a square or circle centered around the vertex of the point. Each vertex has an associated point size that controls the size of that square or circle.

If no vertex or geometry shader is active, the size of the point is controlled by

```
      void PointSize(float size);
```

<size> specifies the requested size of a point. The default value is 1.0. A value less than or equal to zero results in the error INVALID_VALUE.

The requested point size is multiplied with a distance attenuation factor, clamped to a specified point size range, and further clamped to the implementation-dependent point size range to produce the derived point size:

```
       derived size = clamp(size * sqrt(1/(a+b*d+c*d^2)))
```

where d is the eye-coordinate distance from the eye, (0,0,0,1) in eye coordinates, to the vertex, and a, b, and c are distance attenuation function coefficients.

If a vertex or geometry shader is active, the derived size depends on the per-vertex point size mode enable. Per-vertex point size mode is enabled or disabled by calling Enable or Disable with the symbolic value PROGRAM_POINT_SIZE_EXT. If per-vertex point size is enabled and a geometry shader is active, the derived point size is taken from the (potentially clipped) point size variable gl_PointSize written by the geometry shader. If per-vertex point size is enabled and no geometry shader is active, the derived point size is taken from the (potentially clipped) point size variable gl_PointSize written by the vertex shader. If per-vertex point size is disabled and a geometry and/or vertex shader is active, the derived point size is taken from the <size> value provided to PointSize, with no distance attenuation applied. In all cases, the derived point size is clamped to the implementation-dependent point size range.

If multisampling is not enabled, the derived size is passed on to rasterization as the point width. ...

Modify section 3.10 "Fog", p. 191

Modify the third paragraph of this section as follows.

If a vertex or geometry shader is active, or if the fog source, as defined below, is FOG_COORD, then c is the interpolated value of the fog coordinate for this fragment. Otherwise, ...

Additions to Chapter 4 of the OpenGL 2.0 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 2.0 Specification (Special Functions)

Change section 5.4 Display Lists, p. 237

Add the command ProgramParameteriEXT to the list of commands that are not compiled into a display list, but executed immediately, under "Program and Shader Objects", p. 241

Additions to Chapter 6 of the OpenGL 2.0 Specification (State and State Requests)

Modify section 6.1.14, Shader and Program Objects, p. 256

Add to the second paragraph on p. 257:

... if <shader> is a fragment shader object, and GEOMETRY_SHADER_EXT is returned if <shader> is a geometry shader object.

Add to the end of the description of GetProgramiv, p. 257:

If <pname> is GEOMETRY_VERTICES_OUT_EXT, the current value of the maximum number of vertices the geometry shader will output is returned. If <pname> is GEOMETRY_INPUT_TYPE_EXT, the current geometry shader input type is returned and can be one of POINTS, LINES, LINES_ADJACENCY_EXT, TRIANGLES or TRIANGLES_ADJACENCY_EXT. If <pname> is GEOMETRY_OUTPUT_TYPE_EXT, the current geometry shader output type is returned and can be one of POINTS, LINE_STRIP or TRIANGLE_STRIP.

Additions to Appendix A of the OpenGL 2.0 Specification (Invariance)

None.

Additions to the AGL/GLX/WGL Specifications

None.

Dependencies on NV_primitive_restart

The spec describes the behavior that primitive restart does not affect the primitive ID counter gl_PrimitiveIDIn. If NV_primitive_restart is not supported, references to that extension in the discussion of the primitive ID should be removed.

Dependencies on EXT_framebuffer_object

If EXT_framebuffer_object (or similar functionality) is not supported, the gl_Layer output has no effect. "FramebufferTextureEXT" and "FramebufferTextureLayerEXT" should be removed from "New Procedures and Functions", and FRAMEBUFFER_ATTACHMENT_LAYERED_EXT, FRAMEBUFFER_INCOMPLETE_LAYER_TARGETS_EXT, and FRAMEBUFFER_INCOMPLETE_LAYER_COUNT_EXT should be removed from "New Tokens".

Otherwise, this extension modifies EXT_framebuffer_object to add the notion of layered framebuffer attachments and framebuffers that can be used in conjunction with geometry shaders to allow programs to direct primitives to a face of a cube map or layer of a three-dimensional texture or one- or two-dimensional array texture. The layer used for rendering can be selected by the geometry shader at run time.

(insert before the end of Section 4.4.2, Attaching Images to Framebuffer Objects)

There are several types of framebuffer-attachable images:

* the image of a renderbuffer object, which is always two-dimensional,

* a single level of a one-dimensional texture, which is treated as a two-dimensional image with a height of one,

* a single level of a two-dimensional or rectangle texture,

* a single face of a cube map texture level, which is treated as a two-dimensional image, or

* a single layer of a one- or two-dimensional array texture or three-dimensional texture, which is treated as a two-dimensional image.

Additionally, an entire level of a three-dimensional texture, cube map texture, or one- or two-dimensional array texture can be attached to an attachment point. Such attachments are treated as an array of two-dimensional images, arranged in layers, and the corresponding attachment point is considered to be layered.

(replace section 4.4.2.3, "Attaching Texture Images to a Framebuffer")

GL supports copying the rendered contents of the framebuffer into the images of a texture object through the use of the routines CopyTexImage{1D|2D}, and CopyTexSubImage{1D|2D|3D}. Additionally, GL supports rendering directly into the images of a texture object.

To render directly into a texture image, a specified level of a texture object can be attached as one of the logical buffers of the currently bound framebuffer object by calling:

```
  void FramebufferTextureEXT(enum target, enum attachment,
                             uint texture, int level);
```

<target> must be FRAMEBUFFER_EXT. <attachment> must be one of the attachment points of the framebuffer listed in table 1.nnn.

If <texture> is zero, any image or array of images attached to the attachment point named by <attachment> is detached, and the state of the attachment point is reset to its initial values. <level> is ignored if <texture> is zero.

If <texture> is non-zero, FramebufferTextureEXT attaches level <level> of the texture object named <texture> to the framebuffer attachment point named by <attachment>. The error INVALID_VALUE is generated if <texture> is not the name of a texture object, or if <level> is not a supported texture level number for textures of the type corresponding to <target>.

The error INVALID_OPERATION is generated if <texture> is the name of a buffer texture.

If <texture> is the name of a three-dimensional texture, cube map texture, or one- or two-dimensional array texture, the texture level attached to the framebuffer attachment point is an array of images, and the framebuffer attachment is considered layered.

The command

```
  void FramebufferTextureLayerEXT(enum target, enum attachment,
                                  uint texture, int level, int layer);
```

operates like FramebufferTextureEXT, except that only a single layer of the texture level, numbered <layer>, is attached to the attachment point. If <texture> is non-zero, the error INVALID_VALUE is generated if <layer> is negative, or if <texture> is not the name of a texture object. The error INVALID_OPERATION is generated unless <texture> is zero or the name of a three-dimensional or one- or two-dimensional array texture.

The command

```
  void FramebufferTextureFaceEXT(enum target, enum attachment,
                                 uint texture, int level, enum face);
```

operates like FramebufferTextureEXT, except that only a single face of a cube map texture, given by <face>, is attached to the attachment point. <face> is one of TEXTURE_CUBE_MAP_POSITIVE_X, TEXTURE_CUBE_MAP_NEGATIVE_X, TEXTURE_CUBE_MAP_POSITIVE_Y, TEXTURE_CUBE_MAP_NEGATIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z, TEXTURE_CUBE_MAP_NEGATIVE_Z. If <texture> is non-zero, the error INVALID_VALUE is generated if <texture> is not the name of a texture object. The error INVALID_OPERATION is generated unless <texture> is zero or the name of a cube map texture.

The command

```
  void FramebufferTexture1DEXT(enum target, enum attachment,
                               enum textarget, uint texture, int level);
```

operates identically to FramebufferTextureEXT, except for two additional restrictions. If <texture> is non-zero, the error INVALID_ENUM is generated if <textarget> is not TEXTURE_1D and the error INVALID_OPERATION is generated unless <texture> is the name of a one-dimensional texture.

The command

```
  void FramebufferTexture2DEXT(enum target, enum attachment,
                               enum textarget, uint texture, int level);
```

operates similarly to FramebufferTextureEXT. If <textarget> is TEXTURE_2D or TEXTURE_RECTANGLE_ARB, <texture> must be zero or the name of a two-dimensional or rectangle texture. If <textarget> is TEXTURE_CUBE_MAP_POSITIVE_X, TEXTURE_CUBE_MAP_NEGATIVE_X, TEXTURE_CUBE_MAP_POSITIVE_Y, TEXTURE_CUBE_MAP_NEGATIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z, or TEXTURE_CUBE_MAP_NEGATIVE_Z, <texture> must be zero or the name of a cube map texture. For cube map textures, only the single face of the cube map texture level given by <textarget> is attached. The error INVALID_ENUM is generated if <texture> is not zero and <textarget> is not one of the values enumerated above. The error INVALID_OPERATION is generated if <texture> is the name of a texture whose type does not match the texture type required by <textarget>.

The command

```
  void FramebufferTexture3DEXT(enum target, enum attachment,
                               enum textarget, uint texture,
                               int level, int zoffset);
```

behaves identically to FramebufferTextureLayerEXT, with the <layer> parameter set to the value of <zoffset>. The error INVALID_ENUM is generated if <textarget> is not TEXTURE_3D. The error INVALID_OPERATION is generated unless <texture> is zero or the name of a three-dimensional texture.

For all FramebufferTexture commands, if <texture> is non-zero and the command does not result in an error, the framebuffer attachment state corresponding to <attachment> is updated based on the new attachment. FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE_EXT is set to TEXTURE, FRAMEBUFFER_ATTACHMENT_OBJECT_NAME_EXT is set to <texture>, and FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL is set to <level>. FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_FACE is set to <textarget> if FramebufferTexture2DEXT is called and <texture> is the name of a cubemap texture; otherwise, it is set to TEXTURE_CUBE_MAP_POSITIVE_X. FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER_EXT is set to <layer> or <zoffset> if FramebufferTextureLayerEXT or FramebufferTexture3DEXT is called; otherwise, it is set to zero. FRAMEBUFFER_ATTACHMENT_LAYERED_EXT is set to TRUE if FramebufferTextureEXT is called and <texture> is the name of a three-dimensional texture, cube map texture, or one- or two-dimensional array texture; otherwise it is set to FALSE.

(modify Section 4.4.4.1, Framebuffer Attachment Completeness -- add to the conditions necessary for attachment completeness)

The framebuffer attachment point <attachment> is said to be "framebuffer attachment complete" if ...:

* If FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE_EXT is TEXTURE and FRAMEBUFFER_ATTACHMENT_OBJECT_NAME_EXT names a three-dimensional texture, FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER_EXT must be smaller than the depth of the texture.

* If FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE_EXT is TEXTURE and FRAMEBUFFER_ATTACHMENT_OBJECT_NAME_EXT names a one- or two-dimensional array texture, FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER_EXT must be smaller than the number of layers in the texture.

(modify section 4.4.4.2, Framebuffer Completeness -- add to the list of conditions necessary for completeness)

* If any framebuffer attachment is layered, all populated attachments must be layered. Additionally, all populated color attachments must be from textures of the same target (i.e., three-dimensional, cube map, or one- or two-dimensional array textures).
  { FRAMEBUFFER_INCOMPLETE_LAYER_TARGETS_EXT }

* If any framebuffer attachment is layered, all attachments must have the same layer count. For three-dimensional textures, the layer count is the depth of the attached volume. For cube map textures, the layer count is always six. For one- and two-dimensional array textures, the layer count is simply the number of layers in the array texture. { FRAMEBUFFER_INCOMPLETE_LAYER_COUNT_EXT }

The enum in { brackets } after each clause of the framebuffer completeness rules specifies the return value of CheckFramebufferStatusEXT (see below) that is generated when that clause is violated. ...

(add section 4.4.7, Layered Framebuffers)

A framebuffer is considered to be layered if it is complete and all of its populated attachments are layered. When rendering to a layered framebuffer, each fragment generated by the GL is assigned a layer number. The layer number for a fragment is zero if

* the fragment is generated by DrawPixels, CopyPixels, or Bitmap,

* geometry shaders are disabled, or

* the current geometry shader does not contain an instruction that statically assigns a value to the built-in output variable gl_Layer.

Otherwise, the layer for each point, line, or triangle emitted by the geometry shader is taken from the layer output of one of the vertices of the primitive. The vertex used is implementation-dependent. To get defined results, all vertices of each primitive emitted should set the same value for gl_Layer. Since the EndPrimitive() built-in function starts a new output primitive, defined results can be achieved if EndPrimitive() is called between two vertices emitted with different layer numbers. A layer number written by a geometry shader has no effect if the framebuffer is not layered.

When fragments are written to a layered framebuffer, the fragment's layer number selects an image from the array of images at each attachment point from which to obtain the destination R, G, B, A values for blending (Section 4.1.8) and to which to write the final color values for that attachment. If the fragment's layer number is negative or greater than the number of layers attached, the effects of the fragment on the framebuffer contents are undefined.

When the Clear command is used to clear a layered framebuffer attachment, all layers of the attachment are cleared.

When commands such as ReadPixels or CopyPixels read from a layered framebuffer, the image at layer zero of the selected attachment is always used to obtain pixel values.

When cube map texture levels are attached to a layered framebuffer, there are six layers attached, numbered zero through five. Each layer number is mapped to a cube map face, as indicated in Table X.4.

layer number	cube map face
0	TEXTURE_CUBE_MAP_POSITIVE_X
1	TEXTURE_CUBE_MAP_NEGATIVE_X
2	TEXTURE_CUBE_MAP_POSITIVE_Y
3	TEXTURE_CUBE_MAP_NEGATIVE_Y
4	TEXTURE_CUBE_MAP_POSITIVE_Z
5	TEXTURE_CUBE_MAP_NEGATIVE_Z

Table X.4, Layer numbers for cube map texture faces. The layers are numbered in the same sequence as the cube map face token values.

(modify Section 6.1.3, Enumerated Queries -- Modify/add to list of <pname> values for GetFramebufferAttachmentParameterivEXT if FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE_EXT is TEXTURE)

If <pname> is FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER_EXT and the attached image is a layer of a three-dimensional texture or one- or two-dimensional array texture, then <params> will contain the specified layer number. Otherwise, <params> will contain the value zero.

If <pname> is FRAMEBUFFER_ATTACHMENT_LAYERED_EXT, then <params> will contain TRUE if an entire level of a three-dimesional texture, cube map texture, or one- or two-dimensional array texture is attached to the <attachment>. Otherwise, <params> will contain FALSE.

(Modify the Additions to Chapter 5, section 5.4)

Add the commands FramebufferTextureEXT, FramebufferTextureLayerEXT, and FramebufferTextureFaceEXT to the list of commands that are not compiled into a display list, but executed immediately.

Dependencies on EXT_framebuffer_blit

If EXT_framebuffer_blit is supported, the EXT_framebuffer_object language should be further amended so that <target> values passed to FramebufferTextureEXT and FramebufferTextureLayerEXT can be DRAW_FRAMEBUFFER_EXT or READ_FRAMEBUFFER_EXT, and that those functions set/query state for the draw framebuffer if <target> is FRAMEBUFFER_EXT.

Dependencies on EXT_texture_array

If EXT_texture_array is not supported, the discussion array textures the layered rendering edits to EXT_framebuffer_object should be removed. Layered rendering to cube map and 3D textures would still be supported.

If EXT_texture_array is supported, the edits to EXT_framebuffer_object supersede those made in EXT_texture_array, except for language pertaining to mipmap generation of array textures.

There are no functional incompatibilities between the FBO support in these two specifications. The only differences are that this extension supports layered rendering and also rewrites certain sections of the core FBO specification more aggressively.

Dependencies on ARB_texture_rectangle

    If ARB_texture_rectangle is not supported, all references to rectangle
    textures in the EXT_framebuffer_object spec language should be removed.

Dependencies on EXT_texture_buffer_object

    If EXT_buffer_object is not supported, the reference to an
    INVALID_OPERATION error if a buffer texture is passed to
    FramebufferTextureEXT should be removed.

GLX protocol

    TBD

Errors

    The error INVALID_VALUE is generated by ProgramParameteriEXT if <pname> is
    GEOMETRY_INPUT_TYPE_EXT and <value> is not one of POINTS, LINES,
    LINES_ADJACENCY_EXT, TRIANGLES or TRIANGLES_ADJACENCY_EXT.

    The error INVALID_VALUE is generated by ProgramParameteriEXT if <pname> is
    GEOMETRY_OUTPUT_TYPE_EXT and <value> is not one of POINTS, LINE_STRIP or
    TRIANGLE_STRIP.

    The error INVALID_VALUE is generated by ProgramParameteriEXT if <pname> is
    GEOMETRY_VERTICES_OUT_EXT and <value> is negative.

    The error INVALID_VALUE is generated by ProgramParameteriEXT if <pname> is
    GEOMETRY_VERTICES_OUT_EXT and <value> exceeds
    MAX_GEOMETRY_OUTPUT_VERTICES_EXT.

    The error INVALID_VALUE is generated by ProgramParameteriEXT if <pname> is
    set to GEOMETRY_VERTICES_OUT_EXT and the product of <value> and the sum of
    all components of all active varying variables exceeds
    MAX_GEOMETRY_TOTAL_OUTPUT_COMPONENTS_EXT.

    The error INVALID_OPERATION is generated if Begin, or any command that
    implicitly calls Begin, is called when a geometry shader is active and:

      * the input primitive type of the current geometry shader is
        POINTS and <mode> is not POINTS,

      * the input primitive type of the current geometry shader is
        LINES and <mode> is not LINES, LINE_STRIP, or LINE_LOOP,

      * the input primitive type of the current geometry shader is
        TRIANGLES and <mode> is not TRIANGLES, TRIANGLE_STRIP or
        TRIANGLE_FAN,

      * the input primitive type of the current geometry shader is
        LINES_ADJACENCY_EXT and <mode> is not LINES_ADJACENCY_EXT or
        LINE_STRIP_ADJACENCY_EXT, or

      * the input primitive type of the current geometry shader is
        TRIANGLES_ADJACENCY_EXT and <mode> is not
        TRIANGLES_ADJACENCY_EXT or TRIANGLE_STRIP_ADJACENCY_EXT.

New State

Get Value	Type	Get Command	Initial Value	Description	Sec.	Attribute
FRAMEBUFFER_ATTACHMENT_ LAYERED_EXT	nxB	GetFramebuffer- Attachment- ParameterivEXT	FALSE	Framebuffer attachment is layered	4.4.2.3	-

Modify the following state value in Table 6.28, Shader Object State, p. 289.

Get Value	Type	Get Command	Value	Description	Sec.	Attribute
SHADER_TYPE	Z2	GetShaderiv	-	Type of shader (vertex, Fragment, geometry)	2.15.1	-

Add the following state to Table 6.29, Program Object State, p. 290

Get Value	Type	Get Command	Initial Value	Description	Sec.	Attribute
GEOMETRY_VERTICES_OUT_EXT	Z+	GetProgramiv	0	max # of output vertices	2.16.4	-
GEOMETRY_INPUT_TYPE_EXT	Z5	GetProgramiv	TRIANGLES	Primitive input type	2.16.1	-
GEOMETRY_OUTPUT_TYPE_EXT	Z3	GetProgramiv	TRIANGLE_ STRIP	Primitive output type	2.16.2	-

New Implementation Dependent State

Get Value	Type	Get Command	Min. Value	Description	Sec.	Attrib
MAX_GEOMETRY_TEXTURE_ IMAGE_UNITS_EXT	Z+	GetIntegerv	16	maximum number of texture image units accessible in a geometry shader	2.16.4	-
MAX_GEOMETRY_OUTPUT_ VERTICES_EXT	Z+	GetIntegerv	256	maximum number of vertices that any geometry shader can can emit	2.16.4	-
MAX_GEOMETRY_TOTAL_ OUTPUT_COMPONENTS_EXT	Z+	GetIntegerv	1024	maximum number of total components (all vertices) of active varyings that a geometry shader can emit	2.16.4	-
MAX_GEOMETRY_UNIFORM_ COMPONENTS_EXT	Z+	GetIntegerv	512	Number of words for geometry shader uniform variables	2.16.3	-
MAX_GEOMETRY_VARYING_ COMPONENTS_EXT	Z+	GetIntegerv	32	Number of components for varying variables between geometry and fragment shaders	2.16.4	-
MAX_VERTEX_VARYING_ COMPONENTS_EXT	Z+	GetIntegerv	32	Number of components for varying variables between Vertex and geometry shaders	2.15.3	-
MAX_VARYING_ COMPONENTS_EXT	Z+	GetIntegerv	32	Alias for MAX_VARYING_FLOATS	2.15.3	-

Modifications to the OpenGL Shading Language Specification version 1.10.59

Including the following line in a shader can be used to control the language features described in this extension:

```
  #extension GL_EXT_geometry_shader4 : <behavior>
```

where <behavior> is as specified in section 3.3.

A new preprocessor #define is added to the OpenGL Shading Language:

```
  #define GL_EXT_geometry_shader4 1
```

Change the introduction to Chapter 2 "Overview of OpenGL Shading" as follows:

The OpenGL Shading Language is actually three closely related languages. These languages are used to create shaders for the programmable processors contained in the OpenGL processing pipeline. The precise definition of these programmable units is left to separate specifications. In this document, we define them only well enough to provide a context for defining these languages. Unless otherwise noted in this paper, a language feature applies to all languages, and common usage will refer to these languages as a single language. The specific languages will be referred to by the name of the processor they target: vertex, geometry or fragment.

Change the last sentence of the first paragraph of section 3.2 "Source Strings" to:

Multiple shaders of the same language (vertex, geometry or fragment) can be linked together to form a single program.

Change the first paragraph of section 4.1.3, "Integers" as follows:

... integers are limited to 16 bits of precision, plus a sign representation in the vertex, geometry and fragment languages..

Change the first paragraph of section 4.1.9, "Arrays", as follows:

Variables of the same type can be aggregated into one- and two-dimensional arrays by declaring a name followed by brackets ([ ] for one-dimensional arrays and [][] for two-dimensional arrays) enclosing an optional size. When an array size is specified in a declaration, it must be an integral constant expression (see Section 4.3.3 "Integral Constant Expressions") greater than zero. If an array is indexed with an expression that is not an integral constant expression or passed as an argument to a function, then its size must be declared before any such use. It is legal to declare an array without a size and then later re-declare the same name as an array of the same type and specify a size. It is illegal to declare an array with a size, and then later (in the same shader) index the same array with an integral constant expression greater than or equal to the declared size. It is also illegal to index an array with a negative constant expression. Arrays declared as formal parameters in a function declaration must specify a size. Undefined behavior results from indexing an array with a non-constant expression that's greater than or equal to the array's size or less than 0. All basic types and structures can be formed into arrays.

Two-dimensional arrays can only be declared as "varying in" variables in a geometry shader. See section 4.3.6 for details. All other declarations of two-dimensional arrays are illegal.

Change the fourth paragraph of section 4.2 "Scoping", as follows:

Shared globals are global variables declared with the same name in independently compiled units (shaders) of the same language (vertex, geometry or fragment) that are linked together .

Change section 4.3 "Type Qualifiers"

Change the "varying", "in" and "out" qualifiers as follows:

varying - linkage between a vertex shader and geometry shader, or between a geometry shader and a fragment shader, or between a vertex shader and a fragment shader.

in - for function parameters passed into a function or for input varying variables (geometry only)

out - for function parameters passed back out of a function, but not initialized for use when passed in. Also for output varying variables (geometry only).

Change section 4.3.6 "Varying" as follows:

Varying variables provide the interface between the vertex shader and geometry shader and also between the geometry shader and fragment shader and the fixed functionality between them. If no geometry shader is present, varying variables also provide the interface between the vertex shader and fragment shader.

The vertex, or geometry shader will compute values per vertex (such as color, texture coordinates, etc) and write them to output variables declared with the "varying" qualifier (vertex or geometry) or "varying out" qualifiers (geometry only). A vertex or geometry shader may also read these output varying variables, getting back the same values it has written. Reading an output varying variable in a vertex or geometry shader returns undefined results if it is read before being written.

A geometry shader may also read from an input varying variable declared with the "varying in" qualifiers. The value read will be the same value as written by the vertex shader for that varying variable. Since a geometry shader operates on primitives, each input varying variable needs to be declared as an array. Each element of such an array corresponds to a vertex of the primitive being processed. If the varying variable is declared as a scalar or matrix in the vertex shader, it will be a one-dimensional array in the geometry shader. Each array can optionally have a size declared. If a size is not specified, it inferred by the linker and depends on the value of the input primitive type. See table 4.3.xxx to determine the exact size. The read-only built-in constant gl_VerticesIn will be set to this value by the linker. If a size is specified, it has to be the size as given by table 4.3.xxx, otherwise a link error will occur. The built-in constant gl_VerticesIn, if so desired, can be used to size the array correctly for each input primitive type. Varying variables can also be declared as arrays in the vertex shader. This means that those, on input to the geometry shader, must be declared as two- dimensional arrays. The first index to the two-dimensional array holds the vertex number. Declaring a size for the first range of the array is optional, just as it is for one-dimensional arrays. The second index holds the per-vertex array data. Declaring a size for the second range of the array is not optional, and has to match the declaration in the vertex shader.

Input primitive type	Value of built-in gl_VerticesIn
POINTS	1
LINES	2
LINES_ADJACENCY_EXT	4
TRIANGLES	3
TRIANGLES_ADJACENCY_EXT	6

Table 4.3.xxxx The value of the built-in variable gl_VerticesIn is determined at link time, based on the input primitive type.

It is illegal to index these varying arrays, or in the case of two-dimensional arrays, the first range of the array, with a negative integral constant expression or an integral constant expression greater than or equal to gl_VerticesIn. A link error will occur in these cases.

Varying variables that are part of the interface to the fragment shader are set per vertex and interpolated in a perspective correct manner, unless flat shaded, over the primitive being rendered. If single-sampling, the interpolated value is for the fragment center. If multi-sampling, the interpolated value can be anywhere within the pixel, including the fragment center or one of the fragment samples.

A fragment shader may read from varying variables and the value read will be the interpolated value, as a function of the fragment's position within the primitive, unless the varying variable is flat shaded. A fragment shader cannot write to a varying variable.

If a geometry shader is present, the type of the varying variables with the same name declared in the vertex shader and the input varying variables in the geometry shader must match, otherwise the link command will fail. Likewise, the type of the output varying variables with the same name declared in the geometry shader and the varying variables in the fragment shader must match.

If a geometry shader is not present, the type of the varying variables with the same name declared in both the vertex and fragment shaders must match, otherwise the link command will fail.

Only those varying variables used (i.e. read) in the geometry or fragment shader must be written to by the vertex or geometry shader; declaring superfluous varying variables in the vertex shader or declaring superfluous output varying variables in the geometry shader is permissible.

Varying variables are declared as in the following example:

```
  varying in float foo[];    // geometry shader input. Size of the
                             // array set as a result of link, based
                             // on the input primitive type.

  varying in float foo[gl_VerticesIn]; // geometry shader input

  varying in float foo[3];   // geometry shader input. Only legal for
                             // the TRIANGLES input primitive type

  varying in float foo[][5]; // Size of the first range set as a
                             // result of link. Each vertex holds an
                             // array of 5 floats.

  varying out vec4 bar;      // geometry output
  varying vec3 normal;       // vertex shader output or fragment
                             // shader input
```

The varying qualifier can be used only with the data types float, vec2, vec3, vec4, mat2, mat3 and mat4 or arrays of these. Structures cannot be varying. Additionally, the "varying in" and "varying out" qualifiers can only be used in a geometry shader.

If no vertex shader is active, the fixed functionality pipeline of OpenGL will compute values for the built-in varying variables that will be consumed by the fragment shader. Similarly, if no fragment shader is active, the vertex shader or geometry shader is responsible for computing and writing to the built-in varying variables that are needed for OpenGL's fixed functionality fragment pipeline.

Varying variables are required to have global scope, and must be declared outside of function bodies, before their first use.

Change section 7.1 "Vertex Shader Special Variables"

Rename this section to "Vertex and Geometry Shader Special Variables"

Anywhere in this section where it reads "vertex language" replace it with "vertex and geometry language".

Anywhere in this section where it reads "vertex shader" replace it with "vertex shader or geometry shader".

Change the second paragraph to:

The variable gl_Position is available only in the vertex and geometry language and is intended for writing the homogeneous vertex position. It can be written at any time during shader execution. It may also be read back by the shader after being written. This value will be used by primitive assembly, clipping, culling, and other fixed functionality operations that operate on primitives after vertex or geometry processing has occurred. Compilers may generate a diagnostic message if they detect gl_Position is read before being written, but not all such cases are detectable. Writing to gl_Position is optional. If gl_Position is not written but subsequent stages of the OpenGL pipeline consume gl_Position, then results are undefined.

Change the last sentence of this section into the following:

The read-only built-in gl_PrimitiveIDIn is available only in the geometry language and is filled with the number of primitives processed by the geometry shader since the last time Begin was called (directly or indirectly via vertex array functions). See section 2.16.4 for more information.

This variable is intrinsically declared as:

```
    int gl_PrimitiveIDIn; // read only
```

The built-in output variable gl_PrimitiveID is available only in the geometry language and provides a single integer that serves as a primitive identifier. This written primitive ID is available to fragment shaders. If a fragment shader using primitive IDs is active and a geometry shader is also active, the geometry shader must write to gl_PrimitiveID or the primitive ID in the fragment shader number is undefined.

The built-in output variable gl_Layer is available only in the geometry language, and provides the number of the layer of textures attached to a FBO to direct rendering to. If a shader statically assigns a value to gl_Layer, layered rendering mode is enabled. See section 2.16.4 for a detailed explanation. If a shader statically assigns a value to gl_Layer, and there is an execution path through the shader that does not set gl_Layer, then the value of gl_Layer may be undefined for executions of the shader that take that path.

These variables area intrinsically declared as:

```
    int gl_PrimitiveID;
    int gl_Layer;
```

These variables can be read back by the shader after writing to them, to retrieve what was written. Reading the variable before writing it results in undefined behavior. If it is written more than once, the last value written is consumed by the subsequent operations.

All built-in variables discussed in this section have global scope.

Change section 7.2 "Fragment Shader Special Variables"

Change the first paragraph on p. 44 as follows:

The fragment shader has access to the read-only built-in variable gl_FrontFacing whose value is true if the fragment belongs to a front-facing primitive. One use of this is to emulate two-sided lighting by selecting one of two colors calculated by the vertex shader or geometry shader.

Change the first sentence of section 7.4 "Built-in Constants"

The following built-in constant is provided to geometry shaders.

```
  const int gl_VerticesIn; // Value set at link time
```

The following built-in constants are provided to the vertex, geometry and fragment shaders:

Change section 7.6 "Varing Variables"

Unlike user-defined varying variables, the built-in varying variables don't have a strict one-to-one correspondence between the vertex language, geometry language and the fragment language. Four sets are provided, one set for the vertex language output, one set for the geometry language output, one set for the fragment language input and another set for the geometry language input. Their relationship is described below.

The following built-in varying variables are available to write to in a vertex shader or geometry shader. A particular one should be written to if any functionality in a corresponding geometry shader or fragment shader or fixed pipeline uses it or state derived from it. Otherwise, behavior is undefined.

Vertex language built-in outputs:

```
  varying vec4 gl_FrontColor;
  varying vec4 gl_BackColor;
  varying vec4 gl_FrontSecondaryColor;
  varying vec4 gl_BackSecondaryColor;
  varying vec4 gl_TexCoord[]; // at most will be gl_MaxTextureCoords
  varying float gl_FogFragCoord;
```

Geometry language built-in outputs:

```
  varying out vec4 gl_FrontColor;
  varying out vec4 gl_BackColor;
  varying out vec4 gl_FrontSecondaryColor;
  varying out vec4 gl_BackSecondaryColor;
  varying out vec4 gl_TexCoord[]; // at most gl_MaxTextureCoords
  varying out float gl_FogFragCoord;
```

For gl_FogFragCoord, the value written will be used as the "c" value on page 160 of the OpenGL 1.4 Specification by the fixed functionality pipeline. For example, if the z-coordinate of the fragment in eye space is desired as "c", then that's what the vertex or geometry shader should write into gl_FogFragCoord.

Indices used to subscript gl_TexCoord must either be an integral constant expressions, or this array must be re-declared by the shader with a size. The size can be at most gl_MaxTextureCoords. Using indexes close to 0 may aid the implementation in preserving varying resources.

The following input varying variables are available to read from in a geometry shader.

```
  varying in vec4 gl_FrontColorIn[gl_VerticesIn];
  varying in vec4 gl_BackColorIn[gl_VerticesIn];
  varying in vec4 gl_FrontSecondaryColorIn[gl_VerticesIn];
  varying in vec4 gl_BackSecondaryColorIn[gl_VerticesIn];
  varying in vec4 gl_TexCoordIn[gl_VerticesIn][]; // at most will be
                                                  // gl_MaxTextureCoords
  varying in float gl_FogFragCoordIn[gl_VerticesIn];
  varying in vec4 gl_PositionIn[gl_VerticesIn];
  varying in float gl_PointSizeIn[gl_VerticesIn];
  varying in vec4 gl_ClipVertexIn[gl_VerticesIn];
```

All built-in variables are one-dimensional arrays, except for gl_TexCoordIn, which is a two-dimensional array. Each element of a one-dimensional array, or the first index of a two-dimensional array, corresponds to a vertex of the primitive being processed and receives their value from the equivalent vertex output varying variables. See also section 4.3.6.

The following varying variables are available to read from in a fragment shader. The gl_Color and gl_SecondaryColor names are the same names as attributes passed to the vertex shader. However, there is no name conflict, because attributes are visible only in vertex shaders and the following are only visible in a fragment shader.

```
  varying vec4 gl_Color;
  varying vec4 gl_SecondaryColor;
  varying vec4 gl_TexCoord[]; // at most will be gl_MaxTextureCoords
  varying float gl_FogFragCoord;
```

The values in gl_Color and gl_SecondaryColor will be derived automatically by the system from gl_FrontColor, gl_BackColor, gl_FrontSecondaryColor, and gl_BackSecondaryColor. This selection process is described in section 2.14.1 of the OpenGL 2.0 Specification. If fixed functionality is used for vertex processing, then gl_FogFragCoord will either be the z-coordinate of the fragment in eye space, or the interpolation of the fog coordinate, as described in section 3.10 of the OpenGL 1.4 Specification. The gl_TexCoord[] values are the interpolated gl_TexCoord[] values from a vertex or geometry shader or the texture coordinates of any fixed pipeline based vertex functionality.

Indices to the fragment shader gl_TexCoord array are as described above in the vertex and geometry shader text.

Change section 8.7 "Texture Lookup Functions"

Change the first paragraph to:

Texture lookup functions are available to vertex, geometry and fragment shaders. However, level of detail is not computed by fixed functionality for vertex or geometry shaders, so there are some differences in operation between texture lookups. The functions.

Change the third and fourth paragraphs to:

In all functions below, the bias parameter is optional for fragment shaders. The bias parameter is not accepted in a vertex or geometry shader. For a fragment shader, if bias is present, it is added to the calculated level of detail prior to performing the texture access operation. If the bias parameter is not provided, then the implementation automatically selects level of detail: For a texture that is not mip-mapped, the texture is used directly. If it is mip- mapped and running in a fragment shader, the LOD computed by the implementation is used to do the texture lookup. If it is mip- mapped and running on the vertex or geometry shader, then the base LOD of the texture is used.

The built-ins suffixed with "Lod" are allowed only in a vertex or geometry shader. For the "Lod" functions, lod is directly used as the level of detail.

Change section 8.9 Noise Functions

Change the first paragraph to:

Noise functions are available to the vertex, geometry and fragment shaders. They are...

Add a section 8.10 Geometry Shader Functions

This section contains functions that are geometry language specific.

Syntax:

```
    void EmitVertex();   // Geometry only
    void EndPrimitive(); // Geometry only
```

Description:

The function EmitVertex() specifies that a vertex is completed. A vertex is added to the current output primitive using the current values of the varying output variables and the current values of the special built-in output variables gl_PointSize, gl_ClipVertex, gl_Layer, gl_Position and gl_PrimitiveID. The values of any unwritten output variables are undefined. The values of all varying output variables and the special built-in output variables are undefined after a call to EmitVertex(). If a geometry shader, in one invocation, emits more vertices than the value GEOMETRY_VERTICES_OUT_EXT, these emits may have no effect.

The function EndPrimitive() specifies that the current output primitive is completed and a new output primitive (of the same type) should be started. This function does not emit a vertex. The effect of EndPrimitive() is roughly equivalent to calling End followed by a new Begin, where the primitive mode is taken from the program object parameter GEOMETRY_OUTPUT_TYPE_EXT. If the output primitive type is POINTS, calling EndPrimitive() is optional.

A geometry shader starts with an output primitive containing no vertices. When a geometry shader terminates, the current output primitive is automatically completed. It is not necessary to call EndPrimitive() if the geometry shader writes only a single primitive.

Add/Change section 9 (Shading language grammar):

```
  init_declarator_list:
    single_declaration
    init_declarator_list COMMA IDENTIFIER
    init_declarator_list COMMA IDENTIFIER array_declarator_suffix
    init_declarator_list COMMA IDENTIFIER EQUAL initializer

  single_declaration:
    fully_specified_type
    fully_specified_type IDENTIFIER
    fully_specified_type IDENTIFIER array_declarator_suffix
    fully_specified_type IDENTIFIER EQUAL initializer

  array_declarator_suffix:
    LEFT_BRACKET RIGHT_BRACKET
    LEFT_BRACKET constant_expression RIGHT_BRACKET
    LEFT_BRACKET RIGHT_BRACKET array_declarator_suffix
    LEFT_BRACKET constant_expression RIGHT_BRACKET
  array_declarator_suffix

  type_qualifier:
    CONST
    ATTRIBUTE     // Vertex only
    VARYING
    VARYING IN    // Geometry only
    VARYING OUT   // Geometry only
    UNIFORM
```

NVIDIA Implementation Details

Because of a hardware limitation, some GeForce 8 series chips use the odd vertex of an incomplete TRIANGLE_STRIP_ADJACENCY_EXT primitive as a replacement adjacency vertex rather than ignoring it.

Issues

1. *How do geometry shaders fit into the existing GL pipeline?*

   RESOLVED: The following diagram illustrates how geometry shaders fit into the "vertex processing" portion of the GL (Chapter 2 of the OpenGL 2.0 Specification).

   First, vertex attributes are specified via immediate-mode commands or through vertex arrays. They can be conventional attributes (e.g., glVertex, glColor, glTexCoord) or generic (numbered) attributes.

   Vertices are then transformed, either using a vertex shader or fixed-function vertex processing. Fixed-function vertex processing includes position transformation (modelview and projection matrices), lighting, texture coordinate generation, and other calculations. The results of either method are a "transformed vertex", which has a position (in clip coordinates), front and back colors, texture coordinates, generic attributes (vertex shader only), and so on. Note that on many current GL implementations, vertex processing is performed by executing a "fixed function vertex shader" generated by the driver.

After vertex transformation, vertices are assembled into primitives, according to the topology (e.g., TRIANGLES, QUAD_STRIP) provided by the call to glBegin(). Primitives are points, lines, triangles, quads, or polygons. Many GL implementations do not directly support quads or polygons, but instead decompose them into triangles as permitted by the spec.

After initial primitive assembly, a geometry shader is executed on each individual point, line, or triangle primitive, if one is active. It can read the attributes of each transformed vertex, perform arbitrary computations, and emit new transformed vertices. These emitted vertices are themselves assembled into primitives according to the output primitive type of the geometry shader.

Then, the colors of the vertices of each primitive are clamped to [0,1] (if color clamping is enabled), and flat shading may be performed by taking the color from the provoking vertex of the primitive.

Each primitive is clipped to the view volume, and to any enabled user-defined clip planes. Color, texture coordinate, and other attribute values are computed for each new vertex introduced by clipping.

After clipping, the position of each vertex (in clip coordinates) is converted to normalized device coordinates in the perspective division (divide by w) step, and to window coordinates in the viewport transformation step.

At the same time, color values may be converted to normalized fixed-point values according to the "Final Color Processing" portion of the specification.

After the vertices of the primitive are transformed to window coordinate, the GL determines if the primitive is front- or back-facing. That information is used for two-sided color selection, where a single set of colors is selected from either the front or back colors associated with each transformed vertex.

When all this is done, the final transformed position, colors (primary and secondary), and other attributes are used for rasterization (Chapter 3 in the OpenGL 2.0 Specification).

When the raster position is specified (via glRasterPos), it goes through the entire vertex processing pipeline as though it were a point. However, geometry shaders are never run on the raster position.

```
                |generic              |conventional
                |vertex               |vertex
                |attributes           |attributes
                |                     |
                | +-------------------+
                | |                   |
                V V                   V
              vertex            fixed-function
              shader              vertex
                 |               processing
                 |                    |
                 |                    |
                 +<-------------------+
                 |                                     Output
                 |position, color,                   Primitive
                 |other vertex data                     Type
                 |                                       |
                 V                                       |
Begin/       primitive        geometry       primitive   |
 End ------> assembly  -----> shader   ----> assembly  <-+
State            |                               |
                 V                               |
                 +<------------------------------+
                 |
                 |
                 |             color            flat
                 +----------> clamping ---->  shading
                 |                               |
                 V                               |
                 +<------------------------------+
                 |
                 |
              clipping
                 |
                 |        perspective       viewport
                 +------>   divide    ----> transform
                 |                              |
                 |                          +---+-----+
                 |                          V         |
                 |           final       facing       |
                 +------>    color     determination  |
                 |         processing       |         |
                 |             |            |         |
                 |             |            |         |
                 |             +-----+ +----+         |
                 |                   | |              |
                 |                   V V              |
                 |                two-sided           |
                 |                 coloring           |
                 |                    |               |
                 |                    |               |
                 +------------------+ | +-------------+
                                    | | |
                                    V V V
                                rasterization
                                      |
                                      |
                                      V
```

2. *Why is this called GL_EXT_geometry_shader4? There aren't any previous versions of this extension, let alone three?*

   RESOLVED: To match its sibling, EXT_gpu_shader4 and the assembly version NV_gpu_program4. This is the fourth generation of shading functionality, hence the "4" in the name.

3. *Should the GL produce errors at Begin time if an application specifies a primitive mode that is "incompatible" with the geometry shader? For example, if the geometry shader operates on triangles and the application sends a POINTS primitive?*

   RESOLVED: Yes. Mismatches of app-specified primitive types and geometry shader input primitive types appear to be errors and would produce weird and wonderful effects.

4. *Can the input primitive type of a geometry shader be determined at run time?*

   RESOLVED: No. Each geometry shader has a single input primitive type, and vertices are presented to the shader in a specific order based on that type.

5. *Can the input primitive type of a geometry shader be changed?*

   DISCUSSION: The input primitive type is a property of the program object. A change of the input primitive type means the program object will need to be re-linked. It would be nice if the input primitive type was known at compile time, so that the compiler can do error checking of the type and the number of vertices being accessed by the shader. Since we allow multiple compilation units to form one geometry shader, it is not clear how to achieve that. Therefore, the input primitive type is a property of the program object, and not of a shader object.

   RESOLVED: Yes, but each change means the program object will have to be re-linked.

6. *Can the output primitive type of a geometry shader be determined at run time?*

   RESOLVED: Not in this extension.

7. *Can the output primitive type of a program object be changed?*

   RESOLVED: Yes, but the program object will have to be re-linked in order for the change to have effect on program execution.

8. *Must the output primitive type of a geometry shader match the input primitive type in any way?*

   RESOLVED: No, you can have a geometry shader generate points out of triangles or triangles out of points. Some combinations are analogous to existing OpenGL operations: reading triangles and writing points or line strips can be used to emulate a subset of PolygonMode functionality. Reading points and writing triangle strips can be used to emulate point sprites.

9. *Are primitives emitted by a geometry shader processed like any other OpenGL primitive?*

   RESOLVED: Yes. Antialiasing, stippling, polygon offset, polygon mode, culling, two-sided lighting and color selection, point sprite operations, and fragment processing all work as expected.

   One limitation is that the only output primitive types supported are points, line strips, and triangle strips, none of which meaningfully support edge flags that are sometimes used in conjunction with the POINT and LINE polygon modes. Edge flags are always ignored for line-mode triangle strips.

10. *Should geometry shaders support additional input primitive types?*

    RESOLVED: Possibly in a future extension. It should be straightforward to build a future extension to support geometry shaders that operate on quads. Other primitive types might be more demanding on hardware. Quads with adjacency would require 12 vertices per shader execution. General polygons may require even more, since there is no fixed bound on the number of vertices in a polygon.

11. *Should geometry shaders support additional output primitive types?*

    RESOLVED: Possibly in a future extension. Additional output types (e.g., independent lines, line loops, triangle fans, polygons) may be useful in the future; triangle fans/polygons seem particularly useful.

12. *How are adjacency primitives processed by the GL?*

    RESOLVED: The primitive type of an adjacent primitive is set as a Begin mode parameter. Any vertex of an adjacency primitive will be treated as a regular vertex, and processed by a vertex shader as well as the geometry shader. The geometry shader cannot output adjacency primitives, thus processing stops with the geometry shader. If a geometry shader is not active, the GL ignores the "adjacent" vertices in the adjacency primitive.

13. *Should we provide additional adjacency primitive types that can be used inside a Begin/End?*

    RESOLVED: Not in this extension. It may be desirable to add new primitive types (e.g., TRIANGLE_FAN_ADJACENCY) in a future extension.

14. *How do geometry shaders interact with RasterPos?*

    RESOLVED: Geometry shaders are ignored when specifying the raster position.

15. *How do geometry shaders interact with pixel primitives (DrawPixels, Bitmap)?*

    RESOLVED: They do not.

16. *Is there a limit on the number of vertices that can be emitted by* a geometry shader?

    RESOLVED: Unfortunately, yes. Besides practical hardware limits, there may also be practical performance advantages when applications guarantee a tight upper bound on the number of vertices a geometry shader will emit. GPUs frequently excecute programs in parallel, and there are substantial implementation challenges to parallel execution of geometry threads that can write an unbounded number of results, particular given that all the primitives generated by the first geometry shader invocation must be consumed before any of the primitives generated by the second program invocation. Limiting the amount of data a geometry shader can write substantially eases the implementation burden.

    A program object, holding a geometry shader, must declare a maximum number of vertices that can be emitted. There is an implementation-dependent limit on the total number of vertices a program object can emit (256 minimum) and the product of the number of vertices emitted and the number of components of all active varying variables (1024 minimum).

    It would be ideal if the limit could be inferred from the instructions in the shader itself, and that would be possible for many shaders, particularly ones with straight-line flow control. For shaders with more complicated flow control (subroutines, data- dependent looping, and so on), it would be impossible to make such an inference and a "safe" limit would have to be used with adverse and possibly unexpected performance consequences.

    The limit on the number of EmitVertex() calls that can be issued can not always be enforced at compile time, or even at Begin time. We specify that if a shader tries to emit more vertices than allowed, emits that exceed the limit may or may not have any effect.

17. *Should it be possible to change the limit GEOMETRY_VERTICES_OUT_EXT, the number of vertices emitted by a geometry shader, after the program object, containing the shader, is linked?*

    RESOLVED: NO. See also issue 31. Changing this limit might require a re-compile and/or re-link of the shaders and program object on certain implementations. Pretending that this limit can be changed without re-linking does not reflect reality.

18. *How do user clipping and geometry shaders interact?*

    RESOLVED: Just like vertex shaders and user clipping interact. The geometry shader needs to provide the (eye) position gl_ClipVertex. Primitives are clipped after geometry shader execution, not before.

19. *How do edge flags interact with adjacency primitives?*

    RESOLVED: If geometry programs are disabled, adjacency primitives are still supported. For TRIANGLES_ADJACENCY_EXT, edge flags will apply as they do for TRIANGLES. Such primitives are rendered as independent triangles as though the adjacency vertices were not provided. Edge flags for the "real" vertices are supported. For all other adjacency primitive types, edge flags are irrelevant.

20. *Now that a third shader object type is added, what combinations of GLSL, assembly (ARB or NV) low level and fixed-function do we want to support?*

    DISCUSSION: With the addition of the geometry shader, the number of combinations the GL pipeline could support doubled (there is no fixed-function geometry shading). Possible combinations now are:

vertex	geometry	fragment
ff/ASM/GLSL	none/ASM/GLSL	ff/ASM/GLSL

    for a total of 3 x 3 x 3 is 27 combinations. Before the geometry shader was added, the number of combinations was 9, and those we need to support. We have a choice on the other 18.

    RESOLUTION: It makes sense to draw a line at raster in the GL pipeline. The 'north' side of this line covers vertex and geometry shaders, the 'south' side fragment shaders. We now add a simple rule that states that if a program object contains anything north of this line, the north side will be 100% GLSL. This means that:

    a) GLSL program objects with a vertex shader can only use a geometry shader and not an assembly geometry program. If an assembly geometry program is enabled, it is bypassed. This also avoids a tricky case -- a GLSL program object with a vertex and a fragment program linked together. Injecting an assembly geometry shader in the middle at run time won't work well.

    b) GLSL program objects with a geometry shader must have a vertex shader (cannot be ARB/NV or fixed-function vertex shading).

    The 'south' side in this program object still can be any of ff/ARB/NV/GLSL.

21. *How do geometry shaders interact with color clamping?*

    RESOLVED: Geometry shader execution occurs prior to color clamping in the pipeline. This means the colors written by vertex shaders are not clamped to [0,1] before they are read by geometry shaders. If color clamping is enabled, any vertex colors written by the geometry shader will have their components clamped to [0,1].

22. *What is a primitive ID and a vertex ID? I am confused.*

    DISCUSSION: A vertex shader can read a built-in attribute that holds the ID of the current vertex it is processing. See the EXT_gpu_shader4 spec for more information on vertex ID. If the geometry shader needs access to a vertex ID as well, it can be passed as a user-defined varying variable. A geometry shader can read a built-in varying variable that holds the ID of the current primitive it is processing. It also has the ability to write to a built-in output primitive ID variable, to communicate the primitive ID to a fragment shader. A fragment shader can read a built-in attribute that holds the ID of the current primitive it is processing. A primitive ID will be generated even if no geometry shader is active.

23. After a call to EmitVertex(), should the values of the output varying variables be retained or be undefined?

DISCUSSION: There is not a clear answer to this question .The underlying HW mechanism is as follows. An array of output registers is set aside to store vertices that make up primitives. After each EmitVertex() a pointer into that array is incremented. The shader no longer has access to the previous set of values. This argues that the values of output varying variables should be undefined after an EmitVertex() call. The shader is responsible for writing values to all varying variables it wants to emit, for each emit. The counter argument to this is that this is not a nice model for GLSL to program in. The compiler can store varying outputs in a temp register and preserve their values across EmitVertex() calls, at the cost of increased register pressure.

RESOLUTION: For now, without being a clear winner, we've decided to go with the undefined option. The shader is responsible for writng values to all varying variabvles it wants to emit, for each emit.

24. How to distinguish between input and output "varying" variables?

DISCUSSION: Geometry shader outputs are varying variables consistent with the existing definition of varying (used to communicate to the fragment processing stage). Geometry inputs are received from a vertex shader writing to its varying variable outputs. The inputs could be called "varying", to match with the vertex shader, or could be called "attributes" to match the vertex shader inputs (which are called attributes).

RESOLUTION: We'll call input variables "varying", and not "attributes". To distinguish between input and output, they will be further qualified with the words "in" and "out" resulting in, for example:

```
varying in float foo;
varying out vec4 bar[];
```

25. What is the syntax for declaring varying input variables?

DISCUSSION: We need a way to distinguish between the vertices of the input primitive. Suggestions:

1. Declare each input varying variable as an unsized array. Its size is inferred by the linker based on the output primitive type.

2. Declare each input varying variable as a sized array. If the size does not match the output primitive type, a link error occurs.

3. Have an array of structures, where the structure contains the attributes for each vertex.

RESOLUTION: Option 1 seems simple and solves the problem, but it is not a clear winner over the other two. To aid the shader writer in figuring out the size of each array, a new built-in constant, gl_VerticesIn, is defined that holds the number of vertices for the current input primitive type.

26. *Does gl_PointSize, gl_Layer, gl_ClipVertex count agains the MAX_GEOMETRY_VARYING_COMPONENTS limit?*

    RESOLUTION: Core OpenGL 2.0 makes a distinction between varying variables, output from a vertex shader and interpolated over a primitive, and 'special built-in variables' that are outputs, but not interpolated across a primitive. Only varying variables do count against the MAX_VERTEX_VARYING_COMPONENTS limit. gl_PointSize, gl_Layer, gl_ClipVertex and gl_Position are 'special built-in' variables, and therefore should not count against the limit. If HW does need to take components away to support those, that is ok. The actual spec language does mention possible implementation dependencies.

27. *Should writing to gl_Position be optional?*

    DISCUSSION: Before this extensions, the OpenGL Shading Language required that gl_Position be written to in a vertex shader. With the addition of geometry shaders, it is not necessary anymore for a vertex shader to output gl_Position. The geometry shader can do so. With the addition of transform-feedback (see the transform feedback specification) it is not necessary useful for the geometry shader to write out gl_Position either.

    RESOLUTION: Yes, this should be optional.

28. *Should geometry shaders be able to select a layer of a 3D texture, cube map texture, or array texture at run time? If so, how?*

    RESOLVED: See also issue 32. This extension provides a per-vertex output called "gl_Layer", which is an integer specifying the layer to render to. In order to get defined results, the value of gl_Layer needs to be constant for each primitive (point, line or triangle) being emitted by a geometry shader. This layer value is used for all fragments generated by that primitive.

    The EXT_framebuffer_object (FBO) extension is used for rendering to textures, but for cube maps and 3D textures, it only provides the ability to attach a single face or layer of such textures.

    This extension generalizes FBO by creates new entry points to bind an entire texture level (FramebufferTextureEXT) or a single layer of a texture level (FramebufferTextureLayerEXT) or a single face of a level of a cube map texture (FramebufferTextureFaceEXT) to an attachment point. The existing FBO binding functions, FramebufferTexture[123]DEXT are retained, and are defined in terms of the more general new functions.

    The new functions do not have a dimension in the function name or a <textarget> parameter, which can be inferred from the provided texture.

    When an entire texel level of a cube map, 3D, or array texture is attached, that attachment is considered layered. The framebuffer is considered layered if any attachment is layered. When the framebuffer is layered, there are three additional completeness requirements:

  * all attachments must be layered
  * all color attachments must be from textures of identical type
  * all attachments must have the same number of layers

We expect subsequent versions of the FBO spec to relax the requirement that all attachments must have the same width and height, and plan to relax the similar requirement for layer count at that time.

When rendering to a layered framebuffer, layer zero is used unless a geometry shader that writes (statically assings, to be precise) to gl_Layer. When rendering to a non-layered framebuffer, the value of gl_Layer is ignored and the set of single-image attachments are used. When reading from a layered framebuffer (e.g., ReadPixels), layer zero is always used. When clearing a layered framebuffer, all layers are cleared to the corresponding clear values.

Several other approaches were considered, including leveraging existing FBO attachment functions and requiring the use of FramebufferTexture3D with a <zoffset> of zero to make a framebuffer attachment "layerable" (attaching layer zero means that the attachment could be used for either layered- or non- layered rendering). Whether rendering was layered or not could either be inferred from the active geometry shader, or set as a new property of the framebuffer object. There is presently no FramebufferParameter API to set a property of a framebuffer, so it would have been necessary to create new set/query APIs if this approach were chosen.

29. *How should per-vertex point size work with geometry shaders?*

RESOLVED: The value of the existing VERTEX_PROGRAM_POINT_SIZE enable, to control the point size behavior of a vertex shader, does not affect geometry shaders. Specifically, If a geometry shader is active, the point size is taken from the point size output gl_PointSize of the vertex shader, regardless of the value of VERTEX_PROGRAM_POINT_SIZE.

30. *Geometry shaders don't provide a QUADS or generic POLYGON input primitive type. In this extension, what happens if an application provides QUADS, QUAD_STRIP, or POLYGON primitives?*

RESOLVED: Not all vendors supporting this extension were able to accept quads and polygon primitives as input, so such functionality was not provided in this extension. This extension requires that primitives provided to the GL must match the input primitive type of the active geometry shader (if any). QUADS, QUAD_STRIP, and POLYGON primitives are considered not to match any input primitive type, so an INVALID_OPERATION error will result.

The NV_geometry_shader4 extension (built on top of this one) allows applications to provide quads or general polygon primitives to a geometry shader with an input primitive type of TRIANGLES. Such primitives are decomposed into triangles, and a geometry shader is run on each triangle independently.

31. *Geometry shaders provide a limit on the number of vertices that can be emitted. Can this limit be changed at dynamically?*

    RESOLVED: See also issue 17. Not in this extension. This functionality was not provided because it would be an expensive operation on some implementations of this extension. The NV_geometry_shader4 extension (layered on top of this one) does allow applications to change this limit dynamically.

    An application can change the vertex output limit at any time. To allow for the possibility of dynamic changes (as in NV_geometry_shader4) but not require it, a limit change is not guaranteed to take effect unless the program object is re-linked. However, there is no guarantee that such limit changes will not take effect immediately.

32. *See also issue 28. Each vertex emitted by a geometry shader can specify a layer to render to using the output variable "gl_Layer". For LINE_STRIP and TRIANGLE_STRIP output primitive types, which vertex's layer is used?*

    RESOLVED: The vertex from which the layer is extracted is unfortunately undefined. In practice, some implementations of this extension will extract the layer number from the first vertex of the output primitive; others will extract it from the last (provoking) vertex. A future geometry shader extension may choose to define this behavior one way or the other.

    To get portable results, the layer number should be the same for all vertices in any single primitive emitted by the geometry shader. The EndPrimitive() built-in function available in a geometry shader starts a new primitive, and the layer number emitted can be safely changed after EndPrimitive() is called.

33. The grammar allows "varying", "varying out", and "varying in" as type-qualifiers for geometry shaders. What does "varying" without "in" or "out" mean for a geometry shader?

    RESOLVED: The "varying" type qualifier in a geometry shader not followed by "in" or "out" means the same as "varying out".

    This is consistent with the specification saying: "In order to seamlessly be able to insert or remove a geometry shader from a program object, the rules, names and types of the output built-in varying variables and user-defined varying variables are the same as for the vertex shader."

Revision History

```
    Rev.    Date    Author    Changes
    ----  --------  --------  -----------------------------------------
     17   05/22/07  mjk       Clarify that "varying" means the same as
                              "varying out" in a geometry shader.

     16   01/10/07  pbrown    Specify that the total component limit is
                              enforced at LinkProgram time.
     15   12/15/06  pbrown    Documented that the '#extension' token
                              for this extension should begin with "GL_",
                              as apparently called for per convention.

     14      --               Pre-release revisions.
```

Name

    EXT_gpu_program_parameters

Name Strings

    GL_EXT_gpu_program_parameters

Contributors

    Pat Brown
    Haroon Sheikh

Contact

    Pat Brown, NVIDIA Corporation (pbrown 'at' nvidia.com)
    Geoff Stahl, Apple Computer, Inc. (gstahl 'at' apple.com)

Status

    Shipping for GeForce 8 Series (November 2006)

Version

```
    Last Modified Date:         9/27/07
    Revision:                   6
```

Number

    320

Dependencies

    ARB_vertex_program or ARB_fragment_program is required.

    This specification is written against the spec language from the
    ARB_vertex_program extension.

Overview

    This extension provides a new set of procedures to load multiple
    consecutive program environment parameters more efficiently, via a single
    GL call instead of multiple calls. This will reduce the amount of CPU
    overhead involved in loading parameters.

    With the existing ARB_vertex_program and ARB_fragment_program APIs,
    program parameters must be loaded one at a time, via separate calls.
    While the NV_vertex_program extension provides a set of similar functions
    that can be used to load program environment parameters (which are
    equivalent to "program parameters" in NV_vertex_program), no such function
    exists for program local parameters.

New Procedures and Functions

```
    void ProgramEnvParameters4fvEXT(enum target, uint index, sizei count,
                                   const float *params);

    void ProgramLocalParameters4fvEXT(enum target, uint index, sizei count,
                                     const float *params);
```

New Tokens

    None.

Additions to Chapter 2 of the OpenGL 2.0 Specification (OpenGL Operation)

(modify ARB_vertex_program and ARB_fragment_program, add paragraph after introduction of ProgramEnvParameter* calls)

The command

```
      void ProgramEnvParameters4fvEXT(enum target, uint index, sizei count,
                                      const float *params);
```

updates the values of the program environment parameters numbered <index> through <index> + <count> - 1 for the given program target <target>. <params> points to an array of 4*<count> values, where the first four are used to update the program environment parameter numbered <index> and the last four update the program environment parameter numbered <index> + <count> - 1. The error INVALID_VALUE is generated if <count> is less than zero or if the sum of <index> and <count> is greater than the number of program environment parameters supported by <target>.

(modify ARB_vertex_program and ARB_fragment_program, add paragraph after introduction of ProgramLocalParameter* calls)

The command

```
      void ProgramLocalParameters4fvEXT(enum target, uint index, sizei count,
                                        const float *params);
```

updates the values of the program local parameters numbered <index> through <index> + <count> - 1 belonging to the program object currently bound to <target>. <params> points to an array of 4*<count> values, where the first four are used to update the program local parameter numbered <index> and the last four update the program local parameter numbered <index> + <count> - 1. The error INVALID_VALUE is generated if <count> is less than zero or if the sum of <index> and <count> is greater than the number of program local parameters supported by <target>.

Additions to Chapter 3 of the OpenGL 2.0 Specification (Rasterization)

    None.

Additions to Chapter 4 of the OpenGL 2.0 Specification (Per-Fragment Operations and the Frame Buffer)

    None.

Additions to Chapter 5 of the OpenGL 2.0 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 2.0 Specification (State and State Requests)

None.

Additions to Appendix A of the OpenGL 2.0 Specification (Invariance)

None.

Additions to the AGL/GLX/WGL Specifications

None.

GLX Protocol

Four new GL commands are added. The following commands are sent to the server as part of a glXRender request:

```
    ProgramEnvParameters4fvEXT
        2           16+16*n         rendering command length
        2           4281            rendering command opcode
        4           ENUM            target
        4           CARD32          index
        4           CARD32          n
        16*n        FLOAT32         params

    ProgramLocalParameters4fvEXT
        2           16+16*n         rendering command length
        2           4282            rendering command opcode
        4           ENUM            target
        4           CARD32          index
        4           CARD32          n
        16*n        FLOAT32         params
```

Errors

INVALID_VALUE is generated by ProgramEnvParameters4fvEXT or ProgramLocalParameters4fvEXT if <count> is less than zero.

INVALID_VALUE is generated by ProgramEnvParameters4fvEXT if <index> plus <count> is greater than the number of program environment parameters supported by <target>.

INVALID_VALUE is generated by ProgramLocalParameters4fvEXT if <index> plus <count> is greater than the number of program local parameters supported by <target>.

New State

None.

Issues

*(1) Should a set of ProgramEnvParameters*EXT() calls be added, or is using NV_vertex_program's ProgramParameters*NV() sufficient?*

RESOLVED: We should add an ARB-style ProgramEnvParameters*() call for naming consistency. Also ProgramParameters*NV() are not available on all platforms.

(2) Should an equivalent set of calls be added to query multiple program parameters at once?

RESOLVED: No.

(3) Should double-precision versions be supported?

RESOLVED: No. Double-precision parameter values will be converted to single-precision in current driver implementations, anyway.

(4) Why is this spec called "EXT_gpu_program_parameters"?

RESOLVED: The functionality provided by this spec applies to more than one program type. The term "GPU" was used in the extension name to indicate functionality common to all supported program types, which are commonly executed on a GPU.

(5) Is it an error to load multiple parameters with a <count> of zero?

RESOLVED: No. However, it was illegal in versions of the spec prior to 9/27/07. The spec was changed to resolve differences between the shipping implementations from NVIDIA (which did enforce the error) and Apple (which did not). The new behavior is more consistent with the standard OpenGL practice of allowing zero to be passed to GLsizei parameters, and avoids the need for special-case behavior to handle/avoid zero counts in both drivers and applications. Since loading zero program parameters has no actual effect, the only difference between the two behaviors is the update of the GL error state.

Revision History

```
Rev.    Date     Author    Changes
----  --------  --------  -----------------------------------------
6     09/27/07  pbrown    Change the spec to indicate that it's not
                          illegal to load zero parameters, just
                          pointless.

5     11/06/06  mjk       Indicate shipping

4     06/28/06  barthold  Make clear that this spec modifies both
                          ARB_vertex_program and ARB_fragment_program.

3     06/27/06  pbrown    Fix incorrect error language in checking the
                          sum of <index> and <count>, added an issue
                          about the spec name.

2     06/02/06  haroon    Changed to EXT. Added contributors.
```

```
    1     04/24/06  pbrown    Initial revision.
```

Name

    EXT_gpu_shader4

Name Strings

    GL_EXT_gpu_shader4

Contact

    Barthold Lichtenbelt, NVIDIA (blichtenbelt 'at' nvidia.com)
    Pat Brown, NVIDIA (pbrown 'at' nvidia.com)

Status

    Multi vendor extension

    Shipping for GeForce 8 Series (November 2006)

Version

```
    Last Modified Date:         02/04/2008
    Author revision:            12
```

Number

    326

Dependencies

    OpenGL 2.0 is required.

    This extension is written against the OpenGL 2.0 specification and version
    1.10.59 of the OpenGL Shading Language specification.

    This extension trivially interacts with ARB_texture_rectangle.

    This extension trivially interacts with GL_EXT_texture_array.

    This extension trivially interacts with GL_EXT_texture_integer.

    This extension trivially interacts with GL_EXT_geometry_shader4

    This extension trivially interacts with GL_EXT_texture_buffer_object.

    NV_primitive_restart trivially affects the definition of this extension.

    ARB_color_buffer_float affects the definition of this extension.
    EXT_draw_instanced affects the definition of this extension.

Overview

    This extension provides a set of new features to the OpenGL Shading
    Language and related APIs to support capabilities of new hardware. In
    particular, this extension provides the following functionality:

       * New texture lookup functions are provided that allow shaders to

access individual texels using integer coordinates referring to the texel location and level of detail. No filtering is performed. These functions allow applications to use textures as one-, two-, and three-dimensional arrays.

* New texture lookup functions are provided that allow shaders to query the dimensions of a specific level-of-detail image of a texture object.

* New texture lookup functions variants are provided that allow shaders to pass a constant integer vector used to offset the texel locations used during the lookup to assist in custom texture filtering operations.

* New texture lookup functions are provided that allow shaders to access one- and two-dimensional array textures. The second, or third, coordinate is used to select the layer of the array to access.

* New "Grad" texture lookup functions are provided that allow shaders to explicitely pass in derivative values which are used by the GL to compute the level-of-detail when performing a texture lookup.

* A new texture lookup function is provided to access a buffer texture.

* The existing absolute LOD texture lookup functions are no longer restricted to the vertex shader only.

* The ability to specify and use cubemap textures with a DEPTH_COMPONENT internal format. This also enables shadow mapping on cubemaps. The 'q' coordinate is used as the reference value for comparisons. A set of new texture lookup functions is provided to lookup into shadow cubemaps.

* The ability to specify if varying variables are interpolated in a non-perspective correct manner, if they are flat shaded or, if multi-sampling, if centroid sampling should be performed.

* Full signed integer and unsigned integer support in the OpenGL Shading Language:

    - Integers are defined as 32 bit values using two's complement.

    - Unsigned integers and vectors thereof are added.

    - New texture lookup functions are provided that return integer values. These functions are to be used in conjunction with new texture formats whose components are actual integers, rather than integers that encode a floating-point value. To support these lookup functions, new integer and unsigned-integer sampler types are introduced.

    - Integer bitwise operators are now enabled.

    - Several built-in functions and operators now operate on integers or vectors of integers.

```
            - New vertex attribute functions are added that load integer
              attribute data and can be referenced in a vertex shader as
              integer data.

            - New uniform loading commands are added to load unsigned integer
              data.

            - Varying variables can now be (unsigned) integers. If declared
              as such, they have to be flat shaded.

            - Fragment shaders can define their own output variables, and
              declare them to be of type floating-point, integer or unsigned
              integer. These variables are bound to a fragment color index
              with the new API command BindFragDataLocationEXT(), and directed
              to buffers using the existing DrawBuffer or DrawBuffers API
              commands.

        * Added new built-in functions truncate() and round() to the shading
          language.

        * A new built-in variable accessible from within vertex shaders that
          holds the index <i> implicitly passed to ArrayElement to specify the
          vertex. This is called the vertex ID.

        * A new built-in variable accessible from within fragment and geometry
          shaders that hold the index of the currently processed
          primitive. This is called the primitive ID.

    This extension also briefly mentions a new shader type, called a geometry
    shader. A geometry shader is run after vertices are transformed, but
    before clipping. A geometry shader begins with a single primitive (point,
    line, triangle. It can read the attributes of any of the vertices in the
    primitive and use them to generate new primitives. A geometry shader has a
    fixed output primitive type (point, line strip, or triangle strip) and
    emits vertices to define a new primitive. Geometry shaders are discussed
    in detail in the GL_EXT_geometry_shader4 specification.
```

New Procedures and Functions

```
    void VertexAttribI1iEXT(uint index, int x);
    void VertexAttribI2iEXT(uint index, int x, int y);
    void VertexAttribI3iEXT(uint index, int x, int y, int z);
    void VertexAttribI4iEXT(uint index, int x, int y, int z, int w);

    void VertexAttribI1uiEXT(uint index, uint x);
    void VertexAttribI2uiEXT(uint index, uint x, uint y);
    void VertexAttribI3uiEXT(uint index, uint x, uint y, uint z);
    void VertexAttribI4uiEXT(uint index, uint x, uint y, uint z,
                             uint w);

    void VertexAttribI1ivEXT(uint index, const int *v);
    void VertexAttribI2ivEXT(uint index, const int *v);
    void VertexAttribI3ivEXT(uint index, const int *v);
    void VertexAttribI4ivEXT(uint index, const int *v);
    void VertexAttribI1uivEXT(uint index, const uint *v);
    void VertexAttribI2uivEXT(uint index, const uint *v);
    void VertexAttribI3uivEXT(uint index, const uint *v);
    void VertexAttribI4uivEXT(uint index, const uint *v);

    void VertexAttribI4bvEXT(uint index, const byte *v);
    void VertexAttribI4svEXT(uint index, const short *v);
    void VertexAttribI4ubvEXT(uint index, const ubyte *v);
    void VertexAttribI4usvEXT(uint index, const ushort *v);

    void VertexAttribIPointerEXT(uint index, int size, enum type,
                                 sizei stride, const void *pointer);

    void GetVertexAttribIivEXT(uint index, enum pname, int *params);
    void GetVertexAttribIuivEXT(uint index, enum pname,
                                uint *params);

    void Uniform1uiEXT(int location, uint v0);
    void Uniform2uiEXT(int location, uint v0, uint v1);
    void Uniform3uiEXT(int location, uint v0, uint v1, uint v2);
    void Uniform4uiEXT(int location, uint v0, uint v1, uint v2,
                       uint v3);

    void Uniform1uivEXT(int location, sizei count, const uint *value);
    void Uniform2uivEXT(int location, sizei count, const uint *value);
    void Uniform3uivEXT(int location, sizei count, const uint *value);
    void Uniform4uivEXT(int location, sizei count, const uint *value);

    void GetUniformuivEXT(uint program, int location, uint *params);

    void BindFragDataLocationEXT(uint program, uint colorNumber,
                                 const char *name);
    int GetFragDataLocationEXT(uint program, const char *name);
```

New Tokens

    Accepted by the <pname> parameters of GetVertexAttribdv,
    GetVertexAttribfv, GetVertexAttribiv, GetVertexAttribIuivEXT and
    GetVertexAttribIivEXT:

```
      VERTEX_ATTRIB_ARRAY_INTEGER_EXT                    0x88FD
```

Returned by the <type> parameter of GetActiveUniform:

SAMPLER_1D_ARRAY_EXT	0x8DC0
SAMPLER_2D_ARRAY_EXT	0x8DC1
SAMPLER_BUFFER_EXT	0x8DC2
SAMPLER_1D_ARRAY_SHADOW_EXT	0x8DC3
SAMPLER_2D_ARRAY_SHADOW_EXT	0x8DC4
SAMPLER_CUBE_SHADOW_EXT	0x8DC5
UNSIGNED_INT	0x1405
UNSIGNED_INT_VEC2_EXT	0x8DC6
UNSIGNED_INT_VEC3_EXT	0x8DC7
UNSIGNED_INT_VEC4_EXT	0x8DC8
INT_SAMPLER_1D_EXT	0x8DC9
INT_SAMPLER_2D_EXT	0x8DCA
INT_SAMPLER_3D_EXT	0x8DCB
INT_SAMPLER_CUBE_EXT	0x8DCC
INT_SAMPLER_2D_RECT_EXT	0x8DCD
INT_SAMPLER_1D_ARRAY_EXT	0x8DCE
INT_SAMPLER_2D_ARRAY_EXT	0x8DCF
INT_SAMPLER_BUFFER_EXT	0x8DD0
UNSIGNED_INT_SAMPLER_1D_EXT	0x8DD1
UNSIGNED_INT_SAMPLER_2D_EXT	0x8DD2
UNSIGNED_INT_SAMPLER_3D_EXT	0x8DD3
UNSIGNED_INT_SAMPLER_CUBE_EXT	0x8DD4
UNSIGNED_INT_SAMPLER_2D_RECT_EXT	0x8DD5
UNSIGNED_INT_SAMPLER_1D_ARRAY_EXT	0x8DD6
UNSIGNED_INT_SAMPLER_2D_ARRAY_EXT	0x8DD7
UNSIGNED_INT_SAMPLER_BUFFER_EXT	0x8DD8

Accepted by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

MIN_PROGRAM_TEXEL_OFFSET_EXT	0x8904
MAX_PROGRAM_TEXEL_OFFSET_EXT	0x8905

Additions to Chapter 2 of the OpenGL 2.0 Specification (OpenGL Operation)

Modify Section 2.7 "Vertex Specification", p.20

Insert before last paragraph, p.22:

The VertexAttrib* commands described so far should not be used to load data for vertex attributes declared as signed or unsigned integers or vectors thereof in a vertex shader. If they are used to load signed or unsigned integer vertex attributes, the value in those attributes are undefined. Instead use the commands

```
void VertexAttribI[1234]{i,ui}EXT(uint index, T values);
void VertexAttribI[1234]{i,ui}vEXT(uint index, T values);
void VertexAttribI4{b,s,ub,us}vEXT(uint index, T values);
```

to specify fixed-point attributes that are not converted to floating-point. These attributes can be accessed in vertex shaders that declare attributes as signed or unsigned integers or vectors. The VertexAttribI4* commands extend the data passed in to a full signed or unsigned integer. If a VertexAttribI* command is used that does not match the type of the attribute declared in a vertex shader, the values in the attributes are undefined. This means that the unsigned versions of the VertexAttribI* commands need to be used to load data for unsigned integer vertex attributes or vectors, and the signed versions of the VertexAttribI* commands for signed integer vertex attributes or vectors. Note that this also means that the VertexAttribI* commands should not be used to load data for a vertex attribute declared as a float, float vector or matrix, otherwise their values are undefined.

Insert at end of function list, p.24:

```
void VertexAttribIPointerEXT(uint index, int size, enum type,
                             sizei stride, const void *pointer);
```

(modify last paragraph, p.24) The <index> parameter in the VertexAttribPointer and VertexAttribIPointerEXT commands identify the generic vertex attribute array being described. The error INVALID_VALUE is generated if <index> is greater than or equal to MAX_VERTEX_ATTRIBS. Generic attribute arrays with integer <type> arguments can be handled in one of three ways: converted to float by normalizing to [0,1] or [-1,1] as specified in table 2.9, converted directly to float, or left as integers. Data for an array specified by VertexAttribPointer will be converted to floating-point by normalizing if the <normalized> parameter is TRUE, and converted directly to floating-point otherwise. Data for an array specified by VertexAttribIPointerEXT will always be left as integer values.

(modify Table 2.4, p. 25)

Command	Sizes	Integer Handling	Types
VertexPointer	2,3,4	cast	...
NormalPointer	3	normalize	...
ColorPointer	3,4	normalize	...
SecondaryColorPointer	3	normalize	...
IndexPointer	1	cast	...
FogCoordPointer	1	n/a	...
TexCoordPointer	1,2,3,4	cast	...
EdgeFlagPointer	1	integer	...
VertexAttribPointer	1,2,3,4	flag	...
VertexAttribIPointerEXT	1,2,3,4	integer	byte, ubyte, short, ushort, int, uint

Table 2.4: Vertex array sizes (values per vertex) and data types. The "integer handling" column indicates how fixed-point data types are handled: "cast" means that they converted to floating-point directly, "normalize" means that they are converted to floating-point by normalizing to [0,1] (for unsigned types) or [-1,1] (for signed types), "integer" means that they remain as integer values, and "flag" means that either "cast" or "normalized" applies, depending on the setting of the <normalized> flag in VertexAttribPointer.

(modify end of pseudo-code, pp. 27-28)

```
  for (j = 1; j < genericAttributes; j++) {
    if (generic vertex attribute j array enabled) {
      if (generic vertex attribute j array is a pure integer array)
      {
        VertexAttribI[size][type]vEXT(j, generic vertex attribute j
                                         array element i);
      } else if (generic vertex attribute j array normalization
                 flag is set and <type> is not FLOAT or DOUBLE) {
        VertexAttrib[size]N[type]v(j, generic verex attribute j
                                      array element i);
      } else {
        VertexAttrib[size][type]v(j, generic verex attribute j
                                     array element i);
      }
    }
  }

  if (generic vertex attribute 0 array enabled) {
    if (generic vertex attribute 0 array is a pure integer array) {
      VertexAttribI[size][type]vEXT(0, generic verex attribute 0
                                       array element i);
    } else if (generic vertex attribute 0 array normalization flag
               is set and <type> is not FLOAT or DOUBLE) {
     VertexAttrib[size]N[type]v(0, generic verex attribute 0
                                   array element i);
    } else {
      VertexAttrib[size][type]v(0, generic verex attribute 0
                                   array element i);
    }
  }
```

Modify section 2.14.7, "Flatshading", p. 69

Add a new paragraph at the end of the section on p. 70 as follows:

If a vertex or geometry shader is active, the flat shading control described so far applies to the built-in varying variables gl_FrontColor, gl_BackColor, gl_FrontSecondaryColor and gl_BackSecondaryColor. Through the OpenGL Shading Language varying qualifier flat any vertex attribute can be flagged to be flat-shaded. See the OpenGL Shading Language Specification section 4.3.6 for more information.

Modify section 2.14.8, "Color and Associated Data Clipping", p. 71

Add to the end of this section:

For vertex shader varying variables specified to be interpolated without perspective correction (using the noperspective keyword), the value of t used to obtain the varying value associated with P will be adjusted to produce results that vary linearly in screen space.

Modify section 2.15.3, "Shader Variables", page 75

Add the following new return types to the description of GetActiveUniform on p. 81.

```
  SAMPLER_1D_ARRAY_EXT,
  SAMPLER_2D_ARRAY_EXT,
  SAMPLER_1D_ARRAY_SHADOW_EXT,
  SAMPLER_2D_ARRAY_SHADOW_EXT,
  SAMPLER_CUBE_SHADOW_EXT,
  SAMPLER_BUFFER_EXT,

  INT_SAMPLER_1D_EXT,
  INT_SAMPLER_2D_EXT,
  INT_SAMPLER_3D_EXT,
  INT_SAMPLER_CUBE_EXT,
  INT_SAMPLER_2D_RECT_EXT,
  INT_SAMPLER_1D_ARRAY_EXT,
  INT_SAMPLER_2D_ARRAY_EXT,
  INT_SAMPLER_BUFFER_EXT,

  UNSIGNED_INT,
  UNSIGNED_INT_VEC2_EXT,
  UNSIGNED_INT_VEC3_EXT,
  UNSIGNED_INT_VEC4_EXT,
  UNSIGNED_INT_SAMPLER_1D_EXT,
  UNSIGNED_INT_SAMPLER_2D_EXT,
  UNSIGNED_INT_SAMPLER_3D_EXT,
  UNSIGNED_INT_SAMPLER_CUBE_EXT,
  UNSIGNED_INT_SAMPLER_2D_RECT_EXT,
  UNSIGNED_INT_SAMPLER_1D_ARRAY_EXT,
  UNSIGNED_INT_SAMPLER_2D_ARRAY_EXT,
  UNSIGNED_INT_SAMPLER_BUFFER_EXT.
```

Add the following uniform loading command prototypes on p. 81 as follows:

```
  void Uniform{1234}uiEXT(int location, T value);
  void Uniform{1234}uivEXT(int location, sizei count, T value);
```

(add the following paragraph to the description of the above commands)

The Uniform*ui{v} commands will load count sets of one to four unsigned integer values into a uniform location defined as a unsigned integer, an unsigned integer vector, an array of unsigned integers or an array of unsigned integer vectors.

(change the first sentence of the last paragraph as follows)

When loading values for a uniform declared as a Boolean, the Uniform*i{v}, Uniform*ui{v} and Uniform*f{v} set of commands can be used to load boolean values.

Modify section 2.15.4 Shader execution, p. 84.

Add a new section "2.15.4.1 Shader Only Texturing" before the sub-section "Texture Access" on p. 85

This section describes texture functionality that is only accessible through vertex, geometry or fragment shaders. Also refer to the OpenGL Shading Language Specification, section 8.7 and Section 3.8 of the OpenGL 2.0 specification.

Note: For unextended OpenGL 2.0 and the OpenGL Shading Language version 1.20, all supported texture internal formats store unsigned integer values but return floating-point results in the range [0, 1] and are considered unsigned "normalized" integers. The ARB_texture_float extension introduces floating-point internal format where components are both stored and returned as floating-point values, and are not clamped. The EXT_texture_integer extension introduces formats that store either signed or unsigned integer values.

This extension defines additional OpenGL Shading Language texture lookup functions, see section 8.7 of the OpenGL Shading Language, that return either signed or unsigned integer values if the internal format of the texture is signed or unsigned, respectively.

Texel Fetches

The OpenGL Shading Language texel fetch functions provide the ability to extract a single texel from a specified texture image. The integer coordinates passed to the texel fetch functions are used directly as the texel coordinates (i, j, k) into the texture image. This in turn means the texture image is point-sampled (no filtering is performed).

The level of detail accessed is computed by adding the specified level-of-detail parameter <lod> to the base level of the texture, level_base.

The texel fetch functions can not perform depth comparisons or access cube maps. Unlike filtered texel accesses, texel fetches do not support LOD clamping or any texture wrap mode, and require a mipmapped minification filter to access any level of detail other than the base level.

The results of the texel fetch are undefined:

* if the computed LOD is less than the texture's base level (level_base) or greater than the maximum level (level_max),

* if the computed LOD is not the texture's base level and the texture's minification filter is NEAREST or LINEAR,

* if the layer specified for array textures is negative or greater than the number of layers in the array texture,

* if the texel at (i,j,k) coordinates refer to a border texel outside the defined extents of the specified LOD, where

```
        i < -b_s, j < -b_s, k < -b_s,
        i >= w_s - b_s, j >= h_s - b_s, or k >= d_s - b_s,
```

  where the size parameters (w_s, h_s, d_s, and b_s) refer to the width, height, depth, and border size of the image, as in equations 3.15, 3.16, and 3.17, or

. if the texture being accessed is not complete (or cube complete for cubemaps).

Texture Size Query

The OpenGL Shading Language texture size functions provide the ability to query the size of a texture image. The LOD value <lod> passed in as an argument to the texture size functions is added to the level_base of the texture to determine a texture image level. The dimensions of that image level, excluding a possible border, are then returned. If the computed texture image level is outside the range [level_base, level_max], the results are undefined. When querying the size of an array texture, both the dimensions and the layer index are returned. Note that buffer textures do not support mipmapping, therefore the previous lod discussion does not apply to buffer textures

Make the section "Texture Access" a subsection of 2.15.4.1

Modify the first paragraph on p. 86 as follows:

Texture lookups involving textures with depth component data can either return the depth data directly or return the results of a comparison with the R value (see section 3.8.14) used to perform the lookup. The comparison operation is requested in the shader by using any of the shadow sampler and in the texture using the TEXTURE COMPARE MODE parameter. These requests must be consistent; the results of a texture lookup are undefined if:

* The sampler used in a texture lookup function is not one of the shadow sampler types, and the texture object's internal format is DEPTH COMPONENT, and the TEXTURE COMPARE MODE is not NONE.

* The sampler used in a texture lookup function is one of the shadow sampler types, and the texture object's internal format is DEPTH COMPONENT, and the TEXTURE COMPARE MODE is NONE.

* The sampler used in a texture lookup function is one of the shadow sampler types, and the texture object's internal format is not DEPTH COMPONENT.

Add a new section "2.15.4.2 Shader Inputs" before "Position Invariance" on p. 86

Besides having access to vertex attributes and uniform variables, vertex shaders can access the read-only built-in variables gl_VertexID and gl_InstanceID. The gl_VertexID variable holds the integer index <i> implicitly passed to ArrayElement() to specify the vertex. The variable gl_InstanceID holds the integer index of the current primitive in an instanced draw call. See also section 7.1 of the OpenGL Shading Language Specification.

Add a new section "2.15.4.3 Shader Outputs"

A vertex shader can write to built-in as well as user-defined varying variables. These values are expected to be interpolated across the primitive it outputs, unless they are specified to be flat shaded. Refer to section 2.15.3 and the OpenGL Shading Language specification sections 4.3.6, 7.1 and 7.6 for more detail.

The built-in output variables gl_FrontColor, gl_BackColor, gl_FrontSecondaryColor, and gl_BackSecondaryColor hold the front and back colors for the primary and secondary colors for the current vertex.

The built-in output variable gl_TexCoord[] is an array and holds the set of texture coordinates for the current vertex.

The built-in output variable gl_FogFragCoord is used as the "c" value, as described in section 3.10 "Fog" of the OpenGL 2.0 specification.

The built-in special variable gl_Position is intended to hold the homogeneous vertex position. Writing gl_Position is optional.

The built-in special variable gl_ClipVertex holds the vertex coordinate used in the clipping stage, as described in section 2.12 "Clipping" of the OpenGL 2.0 specification.

The built in special variable gl_PointSize, if written, holds the size of the point to be rasterized, measured in pixels.

Number section "Position Invariance", "Validation" and "Undefined Behavior" as sections 2.15.4.4, 2.15.4.5, and 2.15.4.6 respectively.

Additions to Chapter 3 of the OpenGL 2.0 Specification (Rasterization)

Modify Section 3.8.1, Texture Image Specification, p. 150

(modify 4th paragraph, p. 151 -- add cubemaps to the list of texture targets that can be used with DEPTH_COMPONENT textures)

Textures with a base internal format of DEPTH_COMPONENT are supported by texture image specification commands only if <target> is TEXTURE_1D, TEXTURE_2D, TEXTURE_CUBE_MAP, TEXTURE_RECTANGLE_ARB, PROXY_TEXTURE_1D, PROXY_TEXTURE_2D, PROXY_TEXTURE_CUBE_MAP, or PROXY_TEXTURE_RECTANGLE_ARB. Using this format in conjunction with any other target will result in an INVALID_OPERATION error.

Modify Section 3.8.8, Texture Minification:

(replace the last paragraph, p. 171): Let s(x,y) be the function that associates an s texture coordinate with each set of window coordinates (x,y) that lie within a primitive; define t(x,y) and r(x,y) analogously.

```
    Let

          u(x,y) = w_t * s(x,y)
          v(x,y) = h_t * t(x,y)    (3.20a)
          w(x,y) = d_t * r(x,y)

    where w_t, h_t, and d_t are as defined by equations 3.15, 3.16, and 3.17
    with w_s, h_s, and d_s equal to the width, height, and depth of the image
    array whose level is level_base. For a one-dimensional texture, define
    v(x,y) == 0 and w(x,y) == 0; for two-dimensional textures, define w(x,y)
    == 0.

    (start a new paragraph with "For a polygon, rho is given at a fragment
    with window coordinates...", and then continue with the original spec
    text.)

    (replace text starting with the last paragraph on p. 172,
    continuing to the end of p. 174)

    The (u,v,w) coordinates are then modified, as follows:

          u'(x,y) = u(x,y) + offsetu_shader,
          v'(x,y) = v(x,y) + offsetv_shader,
          w'(x,y) = w(x,y) + offsetw_shader

    where (offsetu_shader, offsetv_shader, offsetw_shader) is the texel offset
    specified in the OpenGL Shading Language texture lookup functions that
    support offsets. If the texture function used does not support offsets, or
    for fixed-function texture accesses, all three shader offsets are taken to
    be zero.

    The (u',v',w') coordinates are then further modified according the texture
    wrap modes, as specified in Table X.19, to generate a new set of
    coordinates (u'',v'',w'').
```

```
  TEXTURE_WRAP_S              Coordinate Transformation
  --------------------------  -------------------------------------
  CLAMP                       u'' = clamp(u', 0, w_t-0.5),
                                     if NEAREST filtering,
                                    clamp(u', 0, w_t),
                                     otherwise
  CLAMP_TO_EDGE               u'' = clamp(u', 0.5, w_t-0.5)
  CLAMP_TO_BORDER             u'' = clamp(u', -0.5, w_t+0.5)
  REPEAT                      u'' = clamp(fmod(u', w_t), 0.5, w_t-0.5)
  MIRROR_CLAMP_EXT            u'' = clamp(fabs(u'), 0.5, w_t-0.5),
                                     if NEAREST filtering, or
                                  = clamp(fabs(u'), 0.5, w_t),
                                     otherwise
  MIRROR_CLAMP_TO_EDGE_EXT    u'' = clamp(fabs(u'), 0.5, w_t-0.5)
  MIRROR_CLAMP_TO_BORDER_EXT  u'' = clamp(fabs(u'), 0.5, w_t+0.5)
  MIRRORED_REPEAT             u'' = w_t -
                                    clamp(fabs(w_t - fmod(u', 2*w_t)),
                                         0.5, w_t-0.5)
```

Table X.19: Texel coordinate wrap mode application. clamp(a,b,c) returns b if a<b, c if a>c, and a otherwise. fmod(a,b) returns a-b*floor(a/b), and fabs(a) returns the absolute value of a. For the v and w coordinates, TEXTURE_WRAP_T and h_t, and TEXTURE_WRAP_R and d_t, respectively, are used.

When lambda indicates minification, the value assigned to TEXTURE_MIN_FILTER is used to determine how the texture value for a fragment is selected.

When TEXTURE_MIN_FILTER is NEAREST the texel in the image array of level level_base that is nearest (in Manhattan distance) to (u'',v'',w'') is obtained. The coordinate (i,j,k) is then computed as (floor(u''), floor(v''), floor(w'')).

For a three-dimensional texture, the texel at location (i,j,k) becomes the texture value. For a two-dimensional texture, k is irrelevant, and the texel at location (i,j) becomes the texture value. For a one-dimensional texture, j and k are irrelevant, and the texel at location i becomes the texture value.

If the selected (i,j,k), (i,j), or i location refers to a border texel that satisfies any of the following conditions:

```
      i < -b_s,
      j < -b_s,
      k < -b_s,
      i >= w_l + b_s,
      j >= h_l + b_s, or
      j >= d_l + b_s,
```

then the border values defined by TEXTURE_BORDER_COLOR are used in place of the non-existent texel. If the texture contains color components, the values of TEXTURE_BORDER_COLOR are interpreted as an RGBA color to match the texture's internal format in a manner consistent with table 3.15. If the texture contains depth components, the first component of TEXTURE_BORDER_COLOR is interpreted as a depth value.

When TEXTURE_MIN_FILTER is LINEAR, a 2x2x2 cube of texels in the image array of level level_base is selected. Let:

```
      i_0   = floor(u'' - 0.5),
      j_0   = floor(v'' - 0.5),
      k_0   = floor(w'' - 0.5),
      i_1   = i_0 + 1,
      j_1   = j_0 + 1,
      k_1   = k_0 + 1,
      alpha = frac(u'' - 0.5),
      beta  = frac(v'' - 0.5), and
      gamma = frac(w'' - 0.5),
```

For a three-dimensional texture, the texture value tau is found as...

(replace last paragraph, p.174) For any texel in the equation above that refers to a border texel outside the defined range of the image, the texel value is taken from the texture border color as with NEAREST filtering.

Rename section 3.8.9 "Texture Magnification" to section 3.8.8

modify the first paragraph of section 3.8.8 "Texture Magnification" as follows:

When lambda indicates magnification, the value assigned to TEXTURE_MAG_FILTER determines how the texture value is obtained. There are two possible values for TEXTURE_MAG_FILTER: NEAREST and LINEAR. NEAREST behaves exactly as NEAREST for TEXTURE_MIN_FILTER and LINEAR behaves exactly as LINEAR for TEXTURE_MIN_FILTER, as described in the previous section, including the wrapping calculations. The level-of-detail level_base texture array is always used for magnification.

modify the last paragraph of section 3.8.8, p. 175, as follows:

The rules for NEAREST or LINEAR filtering are then applied to the selected array. Specifically, the coordinate (u,v,w) is computed as in equation 3.20a, with w_s, h_s, and d_s equal to the width, height, and depth of the image array whose level is 'd'.

Modify the second paragraph on p. 176

The rules for NEAREST or LINEAR filtering are then applied to each of the selected arrays, yielding two corresponding texture valutes Tau1 and Tau2. Specifically, for level d1, the coordinate (u,v,w) is computed as in equation 3.20a, with w_s, h_s, and d_s equal to the width, height, and depth of the image array whose level is 'd1'. For level d2 the coordinate (u', v', w') is computed as in equation 3.20a, with w_s, h_s, and d_s equal to the width, height, and depth of the image array whose level is 'd2'.

Modify Section 3.8.14, Texture Comparison Modes (p. 185)

(modify 2nd paragraph, p. 188, indicating that the Q texture coordinate is used for depth comparisons on cubemap textures)

Let D_t be the depth texture value, in the range [0, 1]. For fixed-function texture lookups, let R be the interpolated <r> texture coordinate, clamped to the range [0, 1]. For texture lookups generated by an OpenGL Shading Language lookup function, let R be the reference value for depth comparisons provided in the lookup function, also clamped to [0, 1]. Then the effective texture value L_t, I_t, or A_t is computed as follows:

Modify section 3.11, Fragment Shaders, p. 193

Modify the third paragraph on p. 194 as follows:

Additionally, when a vertex shader is active, it may define one or more varying variables (see section 2.15.3 and the OpenGL Shading Language Specification). These values are, if not flat shaded, interpolated across the primitive being rendered. The results of these interpolations are available when varying variables of the same name are defined in the fragment shader.

Add the following paragraph to the end of section 3.11.1, p. 194

A fragment shader can also write to varying out variables. Values written to these variables are used in the subsequent per-fragment operations. Varying out variables can be used to write floating-point, integer or unsigned integer values destined for buffers attached to a framebuffer object, or destined for color buffers attached to the default framebuffer. The subsection 'Shader Outputs' of the next section describes API how to direct these values to buffers.

Add a new paragraph at the beginning of the section "Texture Access", p. 194

Section 2.15.4.1 describes texture lookup functionality accessible to a vertex shader. The texel fetch and texture size query functionality described there also applies to fragment shaders.

Modify the second paragraph on p. 195 as follows:

Texture lookups involving textures with depth component data can either return the depth data directly or return the results of a comparison with the R value (see section 3.8.14) used to perform the lookup. The comparison operation is requested in the shader by using any of the shadow sampler and in the texture using the TEXTURE COMPARE MODE parameter. These requests must be consistent; the results of a texture lookup are undefined if:

* The sampler used in a texture lookup function is not one of the shadow sampler types, and the texture object's internal format is DEPTH COMPONENT, and the TEXTURE COMPARE MODE is not NONE.

* The sampler used in a texture lookup function is one of the shadow sampler types, and the texture object's internal format is DEPTH COMPONENT, and the TEXTURE COMPARE MODE is NONE.

* The sampler used in a texture lookup function is one of the shadow sampler types, and the texture object's internal format is not DEPTH COMPONENT.

Add the following paragraph to the section Shader Inputs, p. 196

If a geometry shader is active, the built-in variable gl_PrimitiveID contains the ID value emitted by the geometry shader for the provoking vertex. If no geometry shader is active, gl_PrimitiveID is filled with the number of primitives processed by the rasterizer since the last time Begin was called (directly or indirectly via vertex array functions). The first primitive generated after a Begin is numbered zero, and the primitive ID counter is incremented after every individual point, line, or polygon primitive is processed. For polygons drawn in point or line mode, the primitive ID counter is incremented only once, even though multiple points or lines may be drawn. For QUADS and QUAD_STRIP primitives that are decomposed into triangles, the primitive ID is incremented after each complete quad is processed. For POLYGON primitives, the primitive ID counter is undefined. The primitive ID is undefined for fragments generated by DrawPixels or Bitmap. Restarting a primitive topology using the primitive restart index has no effect on the primitive ID counter.

Modify the first paragraph of the section Shader Outputs, p. 196 as follows

The OpenGL Shading Language specification describes the values that may be output by a fragment shader. These outputs are split into two categories. User-defined varying out variables and built-in variables. The built-in variables are gl_FragColor, gl_FragData[n], and gl_FragDepth. If fragment clamping is enabled, the final fragment color values or the final fragment data values or the final varying out variable values written by a fragment shader are clamped to the range [0,1] and then may be converted to fixed-point as described in section 2.14.9. Only user-defined varying out variables declared as a floating-point type are clamped and may be converted. If fragment clamping is disabled, the final fragment color values or the final fragment data values or the final varying output variable values are not modified. The final fragment depth written...

Modify the second paragraph of the section Shader Outputs, p. 196 as follows

...A fragment shader may not statically assign values to more than one of gl_FragColor, gl_FragData or any user-defined varying output variable. In this case, a compile or link error will result. A shader statically...

Add the following to the end of the section Shader Outputs, p. 197

The values of user-defined varying out variables are directed to a color buffer in a two step process. First the varying out variable is bound to a fragment color by using its number. The GL will assign a number to each varying out variable, unless overridden by the command BindFragDataLocationEXT(). The number of the fragment color assigned for each user-defined varying out variable can be queried with GetFragDataLocationEXT(). Next, the DrawBuffer or DrawBuffers commands (see section 4.2.1) direct each fragment color to a particular buffer.

The binding of a user-defined varying out variable to a fragment color number can be specified explicitly. The command

```
    void BindFragDataLocationEXT(uint program, uint colorNumber,
                                 const char *name);
```

specifies that the varying out variable name in program should be bound to fragment color colorNumber when the program is next linked. If name was bound previously, its assigned binding is replaced with colorNumber. name must be a null terminated string. The error INVALID_VALUE is generated if colorNumber is equal or greater than MAX_DRAW_BUFFERS. BindFragDataLocationEXT has no effect until the program is linked. In particular, it doesn't modify the bindings of varying out variables in a program that has already been linked. The error INVALID OPERATION is generated if name starts with the reserved "gl_" prefix.

When a program is linked, any varying out variables without a binding specified through BindFragDataLocationEXT will automatically be bound to fragment colors by the GL. Such bindings can be queried using the command GetFragDataLocationEXT. LinkProgram will fail if the assigned binding of a varying out variable would cause the GL to reference a non-existant fragment color number (one greater than or equal to MAX DRAW_BUFFERS). LinkProgram will also fail if more than one varying out variable is bound to the same number. This type of aliasing is not allowed.

BindFragDataLocationEXT may be issued before any shader objects are attached to a program object. Hence it is allowed to bind any name (except a name starting with "gl_") to a color number, including a name that is never used as a varying out variable in any fragment shader object. Assigned bindings for variables that do not exist are ignored.

After a program object has been linked successfully, the bindings of varying out variable names to color numbers can be queried. The command

```
int GetFragDataLocationEXT(uint program, const char *name);
```

returns the number of the fragment color that the varying out variable name was bound to when the program object program was last linked. name must be a null terminated string. If program has not been successfully linked, the error INVALID OPERATION is generated. If name is not a varying out variable, or if an error occurs, -1 will be returned.

Additions to Chapter 4 of the OpenGL 2.0 Specification (Per-Fragment Operations and the Frame Buffer)

Modify Section 4.2.1, Selecting a Buffer for Writing (p. 212)

(modify next-to-last paragraph, p. 213) If a fragment shader writes to gl_FragColor, DrawBuffers specifies a set of draw buffers into which the single fragment color defined by gl_FragColor is written. If a fragment shader writes to gl_FragData or a user-defined varying out variable, DrawBuffers specifies a set of draw buffers into which each of the multiple output colors defined by these variables are separately written. If a fragment shader writes to neither gl_FragColor, nor gl FragData, nor any user-defined varying out variables, the values of the fragment colors following shader execution are undefined, and may differ for each fragment color.

Additions to Chapter 5 of the OpenGL 2.0 Specification (Special Functions)

Change section 5.4 Display Lists, p. 237

Add the commands VertexAttribIPointerEXT and BindFragDataLocationEXT to the list of commands that are not compiled into a display list, but executed immediately, under "Program and Shader Objects", p. 241

Additions to Chapter 6 of the OpenGL 2.0 Specification (State and State Requests)

Modify section 6.1.14 "Shader and Program Queries", p. 256

Modify 2nd paragraph, p.259:

Add the following to the list of GetVertexAttrib* commands:

```
  void GetVertexAttribIivEXT(uint index, enum pname, int *params);
  void GetVertexAttribIuivEXT(uint index, enum pname, uint *params);
```

obtain the... <pname> must be one of VERTEX_ATTRIB_ARRAY_ENABLED ,., VERTEX_ATTRIB_ARRAY_NORMALIZED, VERTEX_ATTRIB_ARRAY_INTEGER_EXT, or CURRENT_VERTEX_ATTRIB. ...

Split 3rd paragraph, p.259

... The size, stride, type, normalized flag, and unconverted integer flag are set by the commands VertexAttribPointer and VertexAttribIPointerEXT. The normalized flag is always set to FALSE by by VertexAttribIPointerEXT. The unconverted integer flag is always set to FALSE by VertexAttribPointer and TRUE by VertexAttribIPointerEXT.

The query CURRENT_VERTEX_ATTRIB returns the current value for the generic attribute <index>. GetVertexAttribdv and GetVertexAttribfv read and return the current attribute values as floating-point values; GetVertexAttribiv reads them as floating-point values and converts them to integer values; GetVertexAttribIivEXT reads and returns them as integers; GetVertexAttribIuivEXT reads and returns them as unsigned integers. The results of the query are undefined if the current attribute values are read using one data type but were specified using a different one. The error INVALID_OPERATION is generated if <index> is zero.

Change the prototypes in the first paragraph on page 260 as follows:

```
  void GetUniformfv(uint program, int location, float *params);
  void GetUniformiv(uint program, int location, int *params);
  void GetUniformuivEXT(uint program, int location, uint *params);
```

Additions to Appendix A of the OpenGL 2.0 Specification (Invariance)

None.

Additions to the AGL/GLX/WGL Specifications

None.

Interactions with GL_ARB_color_buffer_float

If the GL_ARB_color_buffer_float extension is not supported then any reference to fragment clamping in section 3.11.2 "Shader Execution" needs to be deleted.

Interactions with GL_ARB_texture_rectangle

If the GL_ARB_texture_rectangle extension is not supported then all references to texture lookup functions with 'Rect' in the name need to be deleted.

Interactions with GL_EXT_texture_array

If the GL_EXT_texture_array extension is not supported, all references to one- and two-dimensional array texture sampler types (e.g., sampler1DArray, sampler2DArray) and the texture lookup functions that use them need to be deleted.

Interactions with GL_EXT_geometry_shader4

If the GL_EXT_geometry_shader4 extension is not supported, all references to a geometry shader need to be deleted.

Interactions with GL_NV_primitive_restart

The spec describes the behavior that primitive restart does not affect the primitive ID counter, including for POLYGON primitives (where one could argue that the restart index starts a new primitive without a new Begin to reset the count). If NV_primitive_restart is not supported, references to that extension in the discussion of the primitive ID counter should be removed.

If NV_primitive_restart is supported, index values causing a primitive restart are not considered as specifying an End command, followed by another Begin. Primitive restart is therefore not guaranteed to immediately update material properties when a vertex shader is active. The spec language on p.64 of the OpenGL 2.0 specification says "changes are not guaranteed to update material parameters, defined in table 2.11, until the following End command."

Interactions with EXT_texture_integer

If the EXT_texture_integer spec is not supported, the discussion about this spec in section 2.15.4.1 needs to be removed. All texture lookup functions that return integers or unsigned integers, as discussed in section 8.7 of the OpenGL Shading Language specification, also need to be removed.

Interactions with EXT_texture_buffer_object

If EXT_texture_buffer_object is not supported, references to buffer textures, as well as the texelFetchBuffer and texelSizeBuffer lookup functions and samplerBuffer types, need to be removed.

Interactions with EXT_draw_instanced

If EXT_draw_instanced is not supported, the value of gl_InstanceID is always zero.

Errors

The error INVALID_VALUE is generated by BindFragDataLocationEXT() if colorNumber is equal or greater than MAX_DRAW_BUFFERS.

The error INVALID OPERATION is generated by BindFragDataLocationEXT() if name starts with the reserved "gl_" prefix.

The error INVALID_OPERATOIN is generated by BindFragDataLocationEXT() or GetFragDataLocationEXT if program is not the name of a program object.

The error INVALID_OPERATION is generated by GetFragDataLocationEXT() if program has not been successfully linked.

New State

(add to table 6.7, p. 268)

Get Value	Type	Get Command	Initial Value	Description	Sec.	Attribute
VERTEX_ATTRIB_ARRAY INTEGER_EXT	16+xB	GetVertexAttrib	FALSE	vertex attrib array has unconverted ints	2.8	vertex-array

New Implementation Dependent State

Get Value	Type	Get Command	Minimum Value	Description	Sec.	Attrib
MIN_PROGRAM_TEXEL_OFFSET_EXT	Z	GetIntegerv	-8	minimum texel offset allowed in lookup	2.x.4.4	-
MAX_PROGRAM_TEXEL_OFFSET_EXT	Z	GetIntegerv	+7	maximum texel offset allowed in lookup	2.x.4.4	-

Modifications to The OpenGL Shading Language Specification, Version 1.10.59

Including the following line in a shader can be used to control the language features described in this extension:

```
  #extension GL_EXT_gpu_shader4 : <behavior>
```

where <behavior> is as specified in section 3.3.

A new preprocessor #define is added to the OpenGL Shading Language:

```
  #define GL_EXT_gpu_shader4 1
```

Add to section 3.6 "Keywords"

Add the following keywords:

```
  noperspective, flat, centroid
```

Remove the unsigned keyword from the list of keywords reserved for future use, and add it to the list of keywords.

The following new vector types are added:

```
uvec2, uvec3, uvec4
```

The following new sampler types are added:

```
sampler1DArray, sampler2DArray, sampler1DArrayShadow,
sampler2DArrayShadow, samplerCubeShadow

isampler1D, isampler2D, isampler3D, isamplerCube, isampler2DRect,
isampler1DArray, isampler2DArray

usampler1D, usampler2D, usampler3D, usamplerCube, usampler2DRect,
usampler1DArray, usampler2DArray

samplerBuffer, isamplerBuffer, usamplerBuffer
```

Add to section 4.1 "Basic Types"

Break the table in this section up in several tables. The first table 4.1.1 is named "scalar, vector and matrix data types". It includes the first row through the 'mat4" row.

Add the following to the first section of this table:

unsigned int	An unsigned integer
uvec2	A two-component unsigned integer vector
uvec3	A three-component unsigned integer vector
uvec4	A four-component unsigned integer vector

Break out the sampler types in a separate table, and name that table 4.1.2 "default sampler types". Add the following sampler types to this new table:

sampler1DArray	handle for accessing a 1D array texture
sampler2DArray	handle for accessing a 2D array texture
sampler1DArrayShadow	handle for accessing a 1D array depth texture with comparison
sampler2DArrayShadow	handle for accessing a 2D array depth texture with comparison
samplerBuffer	handle for accessing a buffer texture

Add a table 4.1.3 called "integer sampler types":

isampler1D	handle for accessing an integer 1D texture
isampler2D	handle for accessing an integer 2D texture
isampler3D	handle for accessing an integer 3D texture
isamplerCube	handle for accessing an integer cube map texture
isampler2DRect	handle for accessing an integer rectangle texture
isampler1DArray	handle for accessing an integer 1D array texture
isampler2DArray	handle for accessing an integer 2D array texture
isamplerBuffer	handle for accessing an integer buffer texture

Add a table 4.1.4 called "unsigned integer sampler types":

usampler1D	handle for accessing an unsigned integer 1D texture
usampler2D	handle for accessing an unsigned integer 2D texture
usampler3D	handle for accessing an unsigned integer 3D texture
usamplerCube	handle for accessing an unsigned integer cube map texture
usampler2DRect	handle for accessing an unsigned integer rectangle texture
usampler1DArray	handle for accessing an unsigned integer 1D array texture
usampler2DArray	handle for accessing an unsigned integer 2D array texture
usamplerBuffer	handle for accessing an unsigned integer buffer texture

Change section 4.1.3 "Integers"

Remove the first two paragraphs and replace with the following:

Signed, as well as unsigned integers, are fully supported. Integers hold whole numbers. Integers have at least 32 bits of precision, including a sign bit. Signed integers are stored using a two's complement representation.

Integers are declared and optionally initialized with integer expressions as in the following example:

```
    int i, j = 42;
    unsigned int k = 3u;
```

Literal integer constants can be expressed in decimal (base 10), octal (base 8), or hexadecimal (base 16) as follows.

```
      integer-constant:
            decimal-constant integer-suffix_opt
            octal-constant integer-suffix_opt
            hexadecimal-constant integer-suffix_opt

      integer-suffix:  one of
            u U
```

Change section 4.3 "Type Qualifiers"

Change the "varying" and "out" qualifier as follows:

varying - linkage between a vertex shader and fragment shader, or between a fragment shader and the back end of the OpenGL pipeline.

out - for function parameters passed back out of a function, but not initialized for use when passed in. Also for output varying variables (fragment only).

In the qualifier table, add the following sub-qualifiers under the varying qualifier:

```
  flat varying
  noperspective varying
  centroid varying
```

Change section 4.3.4 "Attribute"

Change the sentence:

The attribute qualifier can be used only with the data types float, vec2, vec3, vec4, mat2, mat3, and mat4.

To:

The attribute qualifier can be used only with the data types int, ivec2, ivec3, ivec4, unsigned int, uvec2, uvec3, uvec4, float, vec2, vec3, vec4, mat2, mat3, and mat4.

Change the fourth paragraph to:

It is expected that graphics hardware will have a small number of fixed locations for passing vertex attributes. Therefore, the OpenGL Shading language defines each non-matrix attribute variable as having space for up to four integer or floating-point values (i.e., a vec4, ivec4 or uvec4). There is an implementation dependent limit on the number of attribute variables that can be used and if this is exceeded it will cause a link error. (Declared attribute variables that are not used do not count against this limit.) A scalar attribute counts the same amount against this limit as a vector of size four, so applications may want to consider packing groups of four unrelated scalar attributes together into a vector to better utilize the capabilities of the underlying hardware. A mat4 attribute will...

Change section 4.3.6 "Varying"

Change the first paragraph to:

Varying variables provide the interface between the vertex shader, the fragment shader, and the fixed functionality between the vertex and fragment shader, as well as the interface from the fragment shader to the back-end of the OpenGL pipeline.

The vertex shader will compute values per vertex (such as color, texture coordinates, etc.) and write them to variables declared with the varying qualifier. A vertex shader may also read varying variables, getting back the same values it has written. Reading a varying variable in a vertex shader returns undefined values if it is read before being written.

The fragment shader will compute values per fragment and write them to variables declared with the varying out qualifier. A fragment shader may also read varying variables, getting back the same result it has written. Reading a varying variable in a fragment shader returns undefined values if it is read before being written.

Varying variables may be written more than once. If so, the last value assigned is the one used.

Change the second paragraph to:

Varying variables that are set per vertex are interpolated by default in a perspective-correct manner over the primitive being rendered, unless the varying is further qualified with noperspective. Interpolation in a perspective correct manner is specified in equations 3.6 and 3.8 in the OpenGL 2.0 specification. When noperspective is specified, interpolation must be linear in screen space, as described in equation 3.7 and the approximation that follows equation 3.8.

If single-sampling, the value is interpolated to the pixel's center, and the centroid qualifier, if present, is ignored. If multi-sampling, and the varying is not qualified with centroid, then the value must be interpolated to the pixel's center, or anywhere within the pixel, or to one of the pixel's samples. If multi-sampling and the varying is qualified with centroid, then the value must be interpolated to a point that lies in both the pixel and in the primitive being rendered, or to one of the pixel's samples that falls within the primitive.

[NOTE: Language for centroid sampling taken from the GLSL 1.20.4 specification]

Varying variables, set per vertex, can be computed on a per-primitive basis (flat shading), or interpolated over a line or polygon primitive (smooth shading). By default, a varying variable is smooth shaded, unless the varying is further qualified with flat. When smooth shading, the varying is interpolated over the primitive. When flat shading, the varying is constant over the primitive, and is taken from the single provoking vertex of the primitive, as described in Section 2.14.7 of the OpenGL 2.0 specification.

Change the fourth paragraph to:

The type and any qualifications (flat, noperspective, centroid) of varying variables with the same name declared in both the vertex and fragment shaders must match, otherwise the link command will fail. Note that built-in varying variables, which have names starting with "gl_", can not be further qualified with flat, noperspective or centroid. The flat keyword cannot be used together with either the noperspective or centroid keywords to further qualify a single varying variable, otherwise a compile error will occur. When using the keywords centroid, flat or noperspective, it must immediately precede the varying keyword. When using both centroid and noperspective keywords, either one can be specified first. Only those varying variables used (i.e. read) in the fragment shader must be written to by the vertex shader; declaring superfluous varying variables in the vertex shader is permissible. Varying out variables, set per fragment, can not be further qualified with flat, noperspective or centroid.

Fragment shaders output values to the back-end of the OpenGL pipeline using either user-defined varying out variables or built-in variables, as described in section 7.2, unless the discard keyword is executed. If the back-end of the OpenGL pipeline consumes a user-defined varying out variable and an execution of a fragment shader does not write a value to that variable, then the value consumed is undefined. If the back-end of the OpenGL pipeline consumes a varying out variable and a fragment shader either writes values into less components of the variable, or if the variable is declared to have less components, than needed, the values of the missing component(s) are undefined. The OpenGL specification, section 3.x.x, describes API to route varying output variables to color buffers.

Add the following examples:

```
  noperspective varying float temperature;
  flat varying vec3 myColor;
  centroid varying vec2 myTexCoord;
  centroid noperspective varying vec2 myTexCoord;
  varying out ivec3 foo;
```

Change the third paragraph on p. 25 as follows:

The "varying" qualifier can be used only with the data types float, vec2, vec3, vec4, mat2, mat3, and mat4, int, ivec2, ivec3, ivec4, unsigned int, uvec2, uvec3, uvec4 or arrays of these. Structures cannot be varying. If the varying is declared as one of the integer or unsigned integer data type variants, then it has to also be qualified as being flat shaded, otherwise a compile error will occur.

The "varying out" qualifier can be used only with the data types float, vec2, vec3, vec4, int, ivec2, ivec3, ivec4, unsigned int, uvec2, uvec3 or uvec4. Structures or arrays cannot be declared as varying out.

Change section 5.1 "Operators"

Remove the "reserved" qualifications from the following operator precedence table entries:

```
  Precedence          Operator class
  ----------          -----------------------------------
      3               (tilde is reserved)
      4               (modulus reserved)
      6               bit-wise shift (reserved)
      9               bit-wise and (reserved)
     10               bit-wise exclusive or (reserved)
     11               bit-wise inclusive or (reserved)
     16               (modulus, shift, and bit-wise are reserved)
```

Change section 5.8 "Assignments"

Change the first bullet from:

   * The arithmetic assignments add into (+=)..

To:

   * The arithmetic assignments add into (+=), subtract from (-
     =), multiply into (*=), and divide into (/=) as well as the
     assignments modulus into (%=), left shift by (<<=), right
     shift by (>>=), and into (&=), inclusive or into (|=),
     exclusive or into (^=). The expression

Delete the last bullet in this paragraph.

Remove the second bullet in the section starting with: The assignments modulus into..

Change section 5.9 "Expressions"

Change the bullet: The operator modulus (%) is reserved for future use to:

* The arithmetic operator % that operates on signed or unsigned integer typed expressions (including vectors). The two operands must be of the same type, or one can be a signed or unsigned integer scalar and the other a signed or unsigned integer vector. If the second operand is zero, results are undefined. If one operand is scalar and the other is a vector, the scalar is applied component-wise to the vector, resulting in the same type as the vector. If both operands are non-negative, then the remainder is non-negative. Results are undefined if one, or both, operands are negative.

Change the last bullet: "Operators and (&), or (|), exclusive or (^), not (~), right-shift (>>), left shift (<<). These operators are reserved for future use." To the following bullets:

* The one's complement operator ~. The operand must be of type signed or unsigned integer (including vectors), and the result is the one's complement of its operand. If the operand is a vector, the operator is applied component-wise to the vector. If the operand is unsigned, the result is computed by subtracting the value from the largest unsigned integer value. If the operand is signed, the result is computed by converting the operand to an unsigned integer, applying ~, and converting back to a signed integer.

* The shift operators << and >>. For both operators, the operands must be of type signed or unsigned integer (including vectors). If the first operand is a scalar, the second operand has to be a scalar as well. The result is undefined if the right operand is negative, or greater than or equal to the number of bits in the left expression's type. The value of E1 << E2 is E1 (interpreted as a bit pattern) left-shifted by E2 bits. The value of E1 >> E2 is E1 right-shifted by E2 bit positions. If E1 is a signed integer, the right-shift will extend the sign bit. If E1 is an unsigned integer, the right-shift will zero-extend.

* The bitwise AND operator &. The operands must be of type signed or unsigned integer (including vectors). The two operands must be of the same type, or one can be a signed or unsigned integer scalar and the other a signed or unsigned integer vector. If one operand is a scalar and the other a vector, the scalar is applied component-wise to the vector, resulting in the same type as the vector. The result is the bitwise AND function of the operands.

* The bitwise exclusive OR operator ^. The operands must be of type signed or unsigned integer (including vectors). The two operands must be of the same type, or one can be a signed or unsigned integer scalar and the other a signed or unsigned integer vector. If one operand is a scalar and the other a vector, the scalar is applied component-wise to the vector, resulting in the same type as the vector. The result is the bitwise exclusive OR function of the operands.

* The bitwise inclusive OR operator |. The operands must be of type signed or unsigned integer (including vectors). The two operands must be of the same type, or one can be a signed or unsigned integer scalar and the other a signed or unsigned integer vector. If one operand is a scalar and the other a vector, the scalar is applied component-wise to the vector, resulting in the same type as the vector. The result is the bitwise inclusive OR function of the operands.

Change Section 7.1 "Vertex Shader Special Variables"

Add the following definition to the list of built-in variable definitions:

```
      int gl_VertexID   // read-only
      int gl_InstanceID // read-only
```

Add the following paragraph at the end of the section:

The variable gl_VertexID is available as a read-only variable from within vertex shaders and holds the integer index <i> implicitly passed to ArrayElement() to specify the vertex. The value of gl_VertexID is defined if and only if:

* the vertex comes from a vertex array command that specifies a complete primitive (e.g. DrawArrays, DrawElements),

* all enabled vertex arrays have non-zero buffer object bindings, and

* the vertex does not come from a display list, even if the display list was compiled using DrawArrays / DrawElements with data sourced from buffer objects.

The variable gl_InstanceID is availale as a read-only variable from within vertex shaders and holds holds the integer index of the current primitive in an instanced draw call (DrawArraysInstancedEXT, DrawElementsInstancedEXT). If the current primitive does not come from an instanced draw call, the value of gl_InstanceID is zero.

Change Section 7.2 "Fragment Shader Special Variables"

Change the 8th and 9th paragraphs on p. 43 as follows:

If a shader statically assigns a value to gl_FragColor, it may not assign a value to any element of gl_FragData nor to any user-defined varying output variable (section 4.3.6). If a shader statically writes a value to any element of gl_FragData, it may not assign a value to gl_FragColor nor to any user-defined varying output variable. That is, a shader may assign values to either gl_FragColor, gl_FragData, or any user-defined varying output variable, but not to a combination of the three options.

If a shader executes the discard keyword, the fragment is discarded, and the values of gl_FragDepth, gl_FragColor, gl_FragData and any user-defined varying output variables become irrelevant.

Add the following paragraph to the top of p. 44:

The variable gl_PrimitiveID is available as a read-only variable from within fragment shaders and holds the id of the currently processed primitive. Section 3.11, subsection "Shader Inputs" of the OpenGL 2.0 specification describes what value it holds based on the primitive type.

Add the following prototype to the list of built-in variables accessible from a fragment shader:

```
int gl_PrimitiveID;
```

Change Chapter 8, sixth paragraph on page 50:

Change the sentence:

When the built-in functions are specified below, where the input arguments (and corresponding output)can be float, vec2, vec3, or vec4, genType is used as the argument.

To:

When the built-in functions are specified below, where the input arguments (and corresponding output) can be float, vec2, vec3, or vec4, genType is used as the argument. Where the input arguments (and corresponding output) can be int, ivec2, ivec3 or ivec4, genIType is used as the argument. Where the input arguments (and corresponding output) can be unsigned int, uvec2, uvec3, or uvec4, genUType is used as the argument.

Add to section 8.3 "Common functions"

Add integer versions of the abs, sign, min, max and clamp functions, as follows:

Syntax:

```
genIType abs(genIType x)

genIType sign(genIType x)

genIType min(genIType x, genIType y)
genIType min(genIType x, int y)
genUType min(genUType x, genUType y)
genUType min(genUType x, unsigned int y)

genIType max(genIType x, genIType y)
genIType max(genIType x, int y)
genUType max(genUType x, genUType y)
genUType max(genUType x, unsigned int y)

genIType clamp(genIType x, genIType minval, genIType maxval)
genIType clamp(genIType x, int minval, int maxval)
genUType clamp(genUType x, genUType minval, genUType maxval)
genUType clamp(genUType x, unsigned int minval,
               unsigned int maxval)
```

Add the following new functions:

Syntax:

```
genType truncate(genType x)
```

Description:

Returns a value equal to the integer closest to x whose absolute value is not larger than the absolute value of x.

Syntax:

```
genType round(genType x)
```

Description:

Returns a value equal to the closest integer to x. If the fractional portion of the operand is 0.5, the nearest even integer is returned. For example, round (1.0) returns 1.0. round(-1.5) returns -2.0. round(3.5) and round (4.5) both return 4.0.

Add to section 8.6 "Vector Relational Functions"

Change the sentence:

Below, "bvec" is a placeholder for one of bvec2, bvec3, or bvec4, "ivec" is a placeholder for one of ivec2, ivec3, or ivec4, and "vec" is a placeholder for vec2, vec3, or vec4.

To:

Below, "bvec" is a placeholder for one of bvec2, bvec3, or bvec4, "ivec" is a placeholder for one of ivec2, ivec3, or ivec4, "uvec" is a placeholder for one of uvec2, uvec3 or uvec4 and "vec" is a placeholder for vec2, vec3, or vec4.

Add uvec versions of all but the any, all and not functions to the table in this section, as follows:

```
bvec lessThan(uvec x, uvec y)
bvec lessThanEqual(uvec x, uvec y)

bvec greaterThan(uvec x, uvec y)
bvec greaterThanEqual(uvec x, uvec y)

bvec equal(uvec x, uvec y)
bvec notEqual(uvec x, uvec y)
```

Add to section 8.7 "Texture Lookup Functions"

Remove the first sentence in the last paragraph:

"The built-ins suffixed with "Lod" are allowed only in a vertex shader.".

Add to this section:

Texture data can be stored by the GL as floating point, unsigned normalized integer, unsigned integer or signed integer data. This is determined by the type of the internal format of the texture. Texture lookups on unsigned normalized integer and floating point data return floating point values in the range [0, 1]. See also section 2.15.4.1 of the OpenGL specification.

Texture lookup functions are provided that can return their result as floating point, unsigned integer or signed integer, depending on the sampler type passed to the lookup function. Care must be taken to use the right sampler type for texture access. Table 8.xxx lists the supported combinations of sampler types and texture internal formats.

texture internal format	default (float) sampler	integer sampler	unsigned integer sampler
float	vec4	n/a	n/a
normalized	vec4	n/a	n/a
signed int	n/a	ivec4	n/a
unsigned int	n/a	n/a	uvec4

Table 8.xxx Valid combinations of the type of the internal format of a texture and the type of the sampler used to access the texture. Each cell in the table indicates the type of the return value of a texture lookup. N/a means this combination is not supported. A texture lookup using a n/a combination will return undefined values. The exceptions to this table are the "textureSize" lookup functions, which will return an integer or integer vector, regardless of the sampler type.

If a texture with a signed integer internal format is accessed, one of the signed integer sampler types must be used. If a texture with an unsigned integer internal format is accessed, one of the unsigned integer sampler types must be used. Otherwise, one of the default (float) sampler types must be used. If the types of a sampler and the corresponding texture internal format do not match, the result of a texture lookup is undefined.

If an integer sampler type is used, the result of a texture lookup is an ivec4. If an unsigned integer sampler type is used, the result of a texture lookup is a uvec4. If a default sampler type is used, the result of a texture lookup is a vec4, where each component is in the range [0, 1].

Integer and unsigned integer functions of all the texture lookup functions described in this section are also provided, except for the "shadow" versions, using function overloading. Their prototypes, however, are not listed separately. These overloaded functions use the integer or unsigned-integer versions of the sampler types and will return an ivec4 or an uvec4 respectively, except for the "textureSize" functions, which will always return an integer, or integer vector. Refer also to table 8.xxxx for valid combinations of texture internal formats and sampler types. For example, for the texture1D function, the complete set of prototypes is:

```
   vec4 texture1D(sampler1D sampler, float coord
                  [, float bias])
   ivec4 texture1D(isampler1D sampler, float coord
                   [, float bias])
   uvec4 texture1D(usampler1D sampler, float coord
                   [, float bias])
```

Add the following new texture lookup functions:

Syntax:

```
  vec4 texelFetch1D(sampler1D sampler, int coord, int lod)
  vec4 texelFetch2D(sampler2D sampler, ivec2 coord, int lod)
  vec4 texelFetch3D(sampler3D sampler, ivec3 coord, int lod)
  vec4 texelFetch2DRect(sampler2DRect sampler, ivec2 coord)
  vec4 texelFetch1DArray(sampler1DArray sampler, ivec2 coord, int lod)
  vec4 texelFetch2DArray(sampler2DArray sampler, ivec3 coord, int lod)
```

Description:

Use integer texture coordinate <coord> to lookup a single texel from the level-of-detail <lod> on the texture bound to <sampler> as described in section 2.15.4.1 of the OpenGL specification "Texel Fetches". For the "array" versions, the layer of the texture array to access is either coord.t or coord.p, depending on the use of the 1D or 2D texel fetch lookup, respectively. Note that texelFetch2DRect does not take a level-of-detail input.

Syntax:

```
  vec4 texelFetchBuffer(samplerBuffer sampler, int coord)
```

Description:

Use integer texture coordinate <coord> to lookup into the buffer texture bound to <sampler>.

Syntax:

```
  int textureSizeBuffer(samplerBuffer sampler)
  int textureSize1D(sampler1D sampler, int lod)
  ivec2 textureSize2D(sampler2D sampler, int lod)
  ivec3 textureSize3D(sampler3D sampler, int lod)
  ivec2 textureSizeCube(samplerCube sampler, int lod)
  ivec2 textureSize2DRect(sampler2DRect sampler, int lod)
  ivec2 textureSize1DArray(sampler1DArray sampler, int lod)
  ivec3 textureSize2DArray(sampler2DArray sampler, int lod)
```

Description:

Returns the dimensions, width, height, depth, and number of layers, of level <lod> for the texture bound to <sampler>, as described in section 2.15.4.1 of the OpenGL specification section "Texture Size Query". For the textureSize1DArray function, the first (".x") component of the returned vector is filled with the width of the texture image and the second component with the number of layers in the texture array. For the textureSize2DArray function, the first two components (".x" and ".y") of

```
    the returned vector are filled with the width and height of the texture
    image respectively. The third component (".z") is filled with the number
    of layers in the texture array.

    Syntax:

      vec4 texture1DArray(sampler1DArray sampler, vec2 coord
                          [, float bias])
      vec4 texture1DArrayLod(sampler1DArray sampler, vec2 coord,
                             float lod)

    Description:

    Use the first element (coord.s) of texture coordinate coord to do a
    texture lookup in the layer indicated by the second coordinate coord.t of
    the 1D texture array currently bound to sampler. The layer to access is
    computed by layer = max (0, min(d - 1, floor (coord.t + 0.5)) where 'd' is
    the depth of the texture array.

    Syntax:

      vec4 texture2DArray(sampler2DArray sampler, vec3 coord
                            [, float bias])
      vec4 texture2DArrayLod(sampler2DArray sampler, vec3 coord,
                               float lod)
    Description:

    Use the first two elements (coord.s, coord.t) of texture coordinate coord
    to do a texture lookup in the layer indicated by the third coordinate
    coord.p of the 2D texture array currently bound to sampler. The layer to
    access is computed by layer = max (0, min(d - 1, floor (coord.p + 0.5))
    where 'd' is the depth of the texture array.

    Syntax:

      vec4 shadow1DArray(sampler1DArrayShadow sampler, vec3 coord,
                         [float bias])
      vec4 shadow1DArrayLod(sampler1DArrayShadow sampler,
                            vec3 coord, float lod)
    Description:

    Use texture coordinate coord.s to do a depth comparison lookup on an array
    layer of the depth texture bound to sampler, as described in section
    3.8.14 of version 2.0 of the OpenGL specification. The layer to access is
    indicated by the second coordinate coord.t and is computed by layer = max
    (0, min(d - 1, floor (coord.t + 0.5)) where 'd' is the depth of the
    texture array. The third component of coord (coord.p) is used as the R
    value. The texture bound to sampler must be a depth texture, or results
    are undefined.

    Syntax:

      vec4 shadow2DArray(sampler2DArrayShadow sampler, vec4 coord)
    Description:

    Use texture coordinate (coord.s, coord.t) to do a depth comparison lookup
    on an array layer of the depth texture bound to sampler, as described in
    section 3.8.14 of version 2.0 of the OpenGL specification. The layer to
    access is indicated by the third coordinate coord.p and is computed by
    layer = max (0, min(d - 1, floor (coord.p + 0.5)) where 'd' is the depth
    of the texture array. The fourth component of coord (coord.q) is used as
    the R value. The texture bound to sampler must be a depth texture, or
    results are undefined.

    Syntax:

        vec4 shadowCube(samplerCubeShadow sampler, vec4 coord)

    Description:

    Use texture coordinate (coord.s, coord.t, coord.p) to do a depth
    comparison lookup on the depth cubemap bound to sampler, as described in
    section 3.8.14. The direction of the vector (coord.s, coord.t, coord.p) is
    used to select which face to do a two-dimensional texture lookup in, as
    described in section 3.8.6 of the OpenGL 2.0 specification. The fourth
    component of coord (coord.q) is used as the R value. The texture bound to
    sampler must be a depth cubemap, otherwise results are undefined.

    Syntax:

      vec4 texture1DGrad(sampler1D sampler, float coord,
                         float ddx, float ddy);
      vec4 texture1DProjGrad(sampler1D sampler, vec2 coord,
                             float ddx, float ddy);
      vec4 texture1DProjGrad(sampler1D sampler, vec4 coord,
                             float ddx, float ddy);
      vec4 texture1DArrayGrad(sampler1DArray sampler, vec2 coord,
                              float ddx, float ddy);

      vec4 texture2DGrad(sampler2D sampler, vec2 coord,
                         vec2 ddx, vec2 ddy);
      vec4 texture2DProjGrad(sampler2D sampler, vec3 coord,
                             vec2 ddx, vec2 ddy);
      vec4 texture2DProjGrad(sampler2D sampler, vec4 coord,
                             vec2 ddx, vec2 ddy);
      vec4 texture2DArrayGrad(sampler2DArray sampler, vec3 coord,
                              vec2 ddx, vec2 ddy);

      vec4 texture3DGrad(sampler3D sampler, vec3 coord,
                         vec3 ddx, vec3 ddy);
      vec4 texture3DProjGrad(sampler3D sampler, vec4 coord,
                             vec3 ddx, vec3 ddy);

      vec4 textureCubeGrad(samplerCube sampler, vec3 coord,
                           vec3 ddx, vec3 ddy);
      vec4 shadow1DGrad(sampler1DShadow sampler, vec3 coord,
                         float ddx, float ddy);
      vec4 shadow1DProjGrad(sampler1DShadow sampler, vec4 coord,
                             float ddx, float ddy);
      vec4 shadow1DArrayGrad(sampler1DArrayShadow sampler, vec3 coord,
                              float ddx, float ddy);

      vec4 shadow2DGrad(sampler2DShadow sampler, vec3 coord,
                         vec2 ddx, vec2 ddy);
      vec4 shadow2DProjGrad(sampler2DShadow sampler, vec4 coord,
                             vec2 ddx, vec2 ddy);
      vec4 shadow2DArrayGrad(sampler2DArrayShadow sampler, vec4 coord,
                              vec2 ddx, vec2 ddy);

      vec4 texture2DRectGrad(sampler2DRect sampler, vec2 coord,
                              vec2 ddx, vec2 ddy);
      vec4 texture2DRectProjGrad(sampler2DRect sampler, vec3 coord,
                                  vec2 ddx, vec2 ddy);
      vec4 texture2DRectProjGrad(sampler2DRect sampler, vec4 coord,
                                  vec2 ddx, vec2 ddy);

      vec4 shadow2DRectGrad(sampler2DRectShadow sampler, vec3 coord,
                             vec2 ddx, vec2 ddy);
      vec4 shadow2DRectProjGrad(sampler2DRectShadow sampler, vec4 coord,
                                 vec2 ddx, vec2 ddy);

      vec4 shadowCubeGrad(samplerCubeShadow sampler, vec4 coord,
                           vec3 ddx, vec3 ddy);
```

Description:

The "Grad" functions map the partial derivatives ddx and ddy to ds/dx, dt/dx, dr/dx, and ds/dy, dt/dy, dr/dy respectively and use texture coordinate "coord" to do a texture lookup as described for their non "Grad" counterparts. The derivatives ddx and ddy are used as the explicit derivate of "coord" with respect to window x and window y respectively and are used to compute lambda_base(x,y) as in equation 3.18 in the OpenGL 2.0 specification. For the "Proj" versions, it is assumed that the partial derivatives ddx and ddy are already projected. I.e. the GL assumes that ddx and ddy represent d(s/q)/dx, d(t/q)/dx, d(r/q)/dx and d(s/q)/dy, d(t/q)/dy, d(r/q)/dy respectively. For the "Cube" versions, the partial derivatives ddx and ddy are assumed to be in the coordinate system used before texture coordinates are projected onto the appropriate cube face. The partial derivatives of the post-projection texture coordinates, which are used for level-of-detail and anisotropic filtering calculations, are derived from coord, ddx and ddy in an implementation-dependent manner.

NOTE: Except for the "array" and shadowCubeGrad() functions, these functions are taken from the ARB_shader_texture_lod spec and are functionally equivalent.

```
    Syntax:

      vec4 texture1DOffset(sampler1D sampler, float coord,
                           int offset [,float bias])
      vec4 texture1DProjOffset(sampler1D sampler, vec2 coord,
                               int offset [,float bias])
      vec4 texture1DProjOffset(sampler1D sampler, vec4 coord,
                               int offset [,float bias])
      vec4 texture1DLodOffset(sampler1D sampler, float coord,
                              float lod, int offset)
      vec4 texture1DProjLodOffset(sampler1D sampler, vec2 coord,
                                  float lod, int offset)
      vec4 texture1DProjLodOffset(sampler1D sampler, vec4 coord,
                                  float lod, int offset)

      vec4 texture2DOffset(sampler2D sampler, vec2 coord,
                           ivec2 offset [,float bias])
      vec4 texture2DProjOffset(sampler2D sampler, vec3 coord,
                               ivec2 offset [,float bias])
      vec4 texture2DProjOffset(sampler2D sampler, vec4 coord,
                               ivec2 offset [,float bias])
      vec4 texture2DLodOffset(sampler2D sampler, vec2 coord,
                              float lod, ivec2 offset)
      vec4 texture2DProjLodOffset(sampler2D sampler, vec3 coord,
                                  float lod, ivec2 offset)
      vec4 texture2DProjLodOffset(sampler2D sampler, vec4 coord,
                                  float lod, ivec2 offset)

      vec4 texture3DOffset(sampler3D sampler, vec3 coord,
                           ivec3 offset [,float bias])
      vec4 texture3DProjOffset(sampler3D sampler, vec4 coord,
                               ivec3 offset [,float bias])
      vec4 texture3DLodOffset(sampler3D sampler, vec3 coord,
                              float lod, ivec3 offset)
      vec4 texture3DProjLodOffset(sampler3D sampler, vec4 coord,
                                  float lod, ivec3 offset)

      vec4 shadow1DOffset(sampler1DShadow sampler, vec3 coord,
                          int offset [,float bias])
      vec4 shadow2DOffset(sampler2DShadow sampler, vec3 coord,
                          ivec2 offset [,float bias])
      vec4 shadow1DProjOffset(sampler1DShadow sampler, vec4 coord,
                              int offset [,float bias])
      vec4 shadow2DProjOffset(sampler2DShadow sampler, vec4 coord,
                              ivec2 offset [,float bias])
      vec4 shadow1DLodOffset(sampler1DShadow sampler, vec3 coord,
                             float lod, int offset)
      vec4 shadow2DLodOffset(sampler2DShadow sampler, vec3 coord,
                             float lod, ivec2 offset)
      vec4 shadow1DProjLodOffset(sampler1DShadow sampler, vec4 coord,
                                 float lod, int offset)
      vec4 shadow2DProjLodOffset(sampler2DShadow sampler, vec4 coord,
                                 float lod, ivec2 offset)
      vec4 texture2DRectOffset(sampler2DRect sampler, vec2 coord,
                               ivec2 offset)
      vec4 texture2DRectProjOffset(sampler2DRect sampler, vec3 coord,
                                   ivec2 offset)
      vec4 texture2DRectProjOffset(sampler2DRect sampler, vec4 coord,
                                   ivec2 offset)
      vec4 shadow2DRectOffset(sampler2DRectShadow sampler, vec3 coord,
                              ivec2 offset)
      vec4 shadow2DRectProjOffset(sampler2DRectShadow sampler, vec4 coord,
                                  ivec2 offset)

      vec4 texelFetch1DOffset(sampler1D sampler, int coord, int lod,
                              int offset)
      vec4 texelFetch2DOffset(sampler2D sampler, ivec2 coord, int lod,
                              ivec2 offset)
      vec4 texelFetch3DOffset(sampler3D sampler, ivec3 coord, int lod,
                              ivec3 offset)
      vec4 texelFetch2DRectOffset(sampler2DRect sampler, ivec2 coord,
                                  ivec2 offset)
      vec4 texelFetch1DArrayOffset(sampler1DArray sampler, ivec2 coord,
                                   int lod, int offset)
      vec4 texelFetch2DArrayOffset(sampler2DArray sampler, ivec3 coord,
                                   int lod, ivec2 offset)

      vec4 texture1DArrayOffset(sampler1DArray sampler, vec2 coord,
                                int offset [, float bias])
      vec4 texture1DArrayLodOffset(sampler1DArray sampler, vec2 coord,
                                   float lod, int offset)

      vec4 texture2DArrayOffset(sampler2DArray sampler, vec3 coord,
                                ivec2 offset [, float bias])
      vec4 texture2DArrayLodOffset(sampler2DArray sampler, vec3 coord,
                                   float lod, ivec2 offset)

      vec4 shadow1DArrayOffset(sampler1DArrayShadow sampler, vec3 coord,
                               int offset, [float bias])
      vec4 shadow1DArrayLodOffset(sampler1DArrayShadow sampler, vec3 coord,
                                  float lod, int offset)

      vec4 shadow2DArrayOffset(sampler2DArrayShadow sampler,
                               vec4 coord, ivec2 offset)

      vec4 texture1DGradOffset(sampler1D sampler, float coord,
                               float ddx, float ddy, int offset);
      vec4 texture1DProjGradOffset(sampler1D sampler, vec2 coord,
                                   float ddx, float ddy, int offset);
      vec4 texture1DProjGradOffset(sampler1D sampler, vec4 coord,
                                   float ddx, float ddy, int offset);
      vec4 texture1DArrayGradOffset(sampler1DArray sampler, vec2 coord,
                                    float ddx, float ddy, int offset);
```

```
    vec4 texture2DGradOffset(sampler2D sampler, vec2 coord,
                             vec2 ddx, vec2 ddy, ivec2 offset);
    vec4 texture2DProjGradOffset(sampler2D sampler, vec3 coord,
                                 vec2 ddx, vec2 ddy, ivec2 offset);
    vec4 texture2DProjGradOffset(sampler2D sampler, vec4 coord,
                                 vec2 ddx, vec2 ddy, ivec2 offset);
    vec4 texture2DArrayGradOffset(sampler2DArray sampler, vec3 coord,
                                  vec2 ddx, vec2 ddy, ivec2 offset);

    vec4 texture3DGradOffset(sampler3D sampler, vec3 coord,
                             vec3 ddx, vec3 ddy, ivec3 offset);
    vec4 texture3DProjGradOffset(sampler3D sampler, vec4 coord,
                                 vec3 ddx, vec3 ddy, ivec3 offset);

    vec4 shadow1DGradOffset(sampler1DShadow sampler, vec3 coord,
                            float ddx, float ddy, int offset);
    vec4 shadow1DProjGradOffset(sampler1DShadow sampler,
                                vec4 coord, float ddx, float ddy,
                                int offset);
    vec4 shadow1DArrayGradOffset(sampler1DArrayShadow sampler,
                                 vec3 coord, float ddx, float ddy,
                                 int offset);

    vec4 shadow2DGradOffset(sampler2DShadow sampler, vec3 coord,
                            vec2 ddx, vec2 ddy, ivec2 offset);
    vec4 shadow2DProjGradOffset(sampler2DShadow sampler, vec4 coord,
                                vec2 ddx, vec2 ddy, ivec2 offset);
    vec4 shadow2DArrayGradOffset(sampler2DArrayShadow sampler,
                                 vec4 coord, vec2 ddx, vec2 ddy,
                                 ivec2 offset);

    vec4 texture2DRectGradOffset(sampler2DRect sampler, vec2 coord,
                                 vec2 ddx, vec2 ddy, ivec2 offset);
    vec4 texture2DRectProjGradOffset(sampler2DRect sampler, vec3 coord,
                                     vec2 ddx, vec2 ddy, ivec2 offset);
    vec4 texture2DRectProjGradOffset(sampler2DRect sampler, vec4 coord,
                                     vec2 ddx, vec2 ddy, ivec2 offset);

    vec4 shadow2DRectGradOffset(sampler2DRectShadow sampler,
                                vec3 coord, vec2 ddx, vec2 ddy,
                                ivec2 offset);
    vec4 shadow2DRectProjGradOffset(sampler2DRectShadow sampler,
                                    vec4 coord, vec2 ddx, vec2 ddy,
                                    ivec2 offset);
```

Description:

The "offset" version of each function provides an extra parameter <offset> which is added to the (u,v,w) texel coordinates before looking up each texel. The offset value must be a constant expression. A limited range of offset values are supported; the minimum and maximum offset values are implementation-dependent and given by MIN_PROGRAM_TEXEL_OFFSET_EXT and MAX_PROGRAM_TEXEL_OFFSET_EXT, respectively. Note that <offset> does not apply to the layer coordinate for texture arrays. This is explained in detail in section 3.8.7 of the OpenGL Specification. Note that texel offsets are also not supported for cubemaps or buffer textures.

Add to section 9 "Grammar"

```
type_qualifer:
    CONST
    ATTRIBUTE  // Vertex only
    varying-modifier_opt VARYING
    UNIFORM

varying-modifier:
    FLAT
    CENTROID
    NOPERSPECTIVE

type_specifier:
    VOID
    FLOAT
    INT
    UNSIGNED_INT
    BOOL
```

Issues

1. *Should we support shorts in GLSL?*

   DISCUSSION:

   RESOLUTION: UNRESOLVED

2. *Do bitwise shifts, AND, exclusive OR and inclusive OR support all combinations of scalars and vectors for each operand?*

   DISCUSSION: It seems sense to support scalar OP scalar, vector OP scalar and vector OP vector. But what about scalar OP vector? Should the scalar be promoted to a vector first?

   RESOLUTION: RESOLVED. Yes, this should work essentially as the '+' operator. The scalar is applied to each component of the vector.

3. *Which built-in functions should also operate on integers?*

   DISCUSSION: There are several that don't make sense to define to operate on integers at all, but the following can be debated: pow, sqrt, dot (and the functions that use dot), cross.

   RESOLUTION: RESOLVED. Integer versions of the abs, sign, min, max and clamp functions are defined. Note that the modulus operator % has been defined for integer operands.

4. *Do we need to support integer matrices?*

   DISCUSSION:

   RESOLUTION: RESOLVED No, not at the moment.

5. Which texture array lookup functions do we need to support?

DISCUSSION: We don't want to support lookup functions that need more than four components passed as parameters. Components can be used for texture coordinates, layer selection, 'R' depth compare and the 'q' coordinate for projection. However, texture projection might be relatively easy to support through code-generation, thus we might be able to support functions that need five components, as long as one of them is 'q' for projective texturing. Specifically, should we support:

```
vec4 texture2DArrayProjLod(sampler2DArray sampler, vec4 coord,
                           float lod)
vec4 shadow1DArray(sampler1DArrayShadow sampler, vec3 coord,
                 [float bias])
vec4 shadow1DArrayProj(sampler1DArrayShadow sampler, vec4 coord,
                       [float bias])
vec4 shadow1DArrayLod(sampler1DArrayShadow sampler, vec3 coord,
                      float lod)
vec4 shadow1DArrayProjLod(sampler1DArrayShadow sampler,
                          vec4 coord, float lod)
vec4 shadow2DArray(sampler2DArrayShadow sampler, vec4 coord)
vec4 shadow2DArrayProj(sampler2DArrayShadow sampler, vec4 coord,
                       float refValue)
```

RESOLUTION: RESOLVED, We'll support all but the "Proj" versions. The assembly spec (NV_gpu_program4) doesn't support the equivalent functionality, either.

6. How do we handle conversions between integer and unsigned integers?

DISCUSSION: Do we allow automatic type conversions between signed and unsigned integers?

RESOLUTION: RESOLVED. We will not add this until GLSL version 1.20 has been defined, and the implicit conversion rules have been established there. If we do this, we would likely only support implicit conversion from int to unsigned int, just like C does.

7. Should varying modifiers (flat, noperspective) apply to built-in varying variables also?

DISCUSSION: There is API to control flat vs smooth shading for colors through glShadeModel(). There is also API to hint if colors should be interpolated perspective correct, or not, through glHint(). These API commands apply to the built-in color varying variables (gl_FrontColor etc). If the varying modifiers in a shader also apply to the color built-ins, which has precedence?

RESOLUTION: RESOLVED. It is simplest and cleanest to only allow the varying modifiers to apply to user-defined varying variables. The behavior of the built-in color varying variables can still be controlled through the API.

8. How should perspective-incorrect interpolation (linear in screen space) and clipping interact?

RESOLVED: Primitives with attributes specified to be perspective-incorrect should be clipped so that the vertices introduced by clipping should have attribute values consistent with the interpolation mode. We do not want to have large color shifts introduced by clipping a perspective-incorrect attribute. For example, a primitive that approaches, but doesn't cross, a frustum clip plane should look pretty much identical to a similar primitive that just barely crosses the clip plane.

Clipping perspective-incorrect interpolants that cross the W==0 plane is very challenging. The attribute clipping equation provided in the spec effectively projects all the original vertices to screen space while ignoring the X and Y frustum clip plane. As W approaches zero, the projected X/Y window coordinates become extremely large. When clipping an edge with one vertex inside the frustum and the other out near infinity (after projection, due to W approaching zero), the interpolated attribute for the entire visible portion of the edge should almost exactly match the attribute value of the visible vertex.

If an outlying vertex approaches and then goes past W==0, it can be said to go "to infinity and beyond" in screen space. The correct answer for screen-linear interpolation is no longer obvious, at least to the author of this specification. Rather than trying to figure out what the "right" answer is or if one even exists, the results of clipping such edges is specified as undefined.

9. Do we need to support a non-MRT fragment shader writing to (unsigned) integer outputs?

DISCUSSION: Fragment shaders with only one fragment output are considered non-MRT shaders. This means that the output of the shader gets smeared across all color buffers attached to the framebuffer. Fragment shaders with multiple fragment outputs are MRT shaders. Each output is directed to a color buffer using the DrawBuffers API (for gl_FragData) and a combination of the BindFragDataLocationEXT and DrawBuffers API (for varying out variables). Before this extension, a non-MRT shader would write to gl_Color only. A shader writing to gl_FragData[] is a MRT shader. With the addition of varying out variables in this extension, any shader writing to a variable out variable is a MRT shader. It is not possible to construct a non-MRT shader writing to varying out variables. Varying out variables can be declared to be of type integer or unsigned integer. In order to support a non-MRT shader that can write to (unsigned) integer outputs, we could define two new built-in variables:

```
  ivec4 gl_FragColorInt;
  uvec4 gl_FragColorUInt;
```

Or we could add a special rule stating that if the program object writes to exactly one varying out variable, it is considered to be non-MRT.

RESOLUTION: NO. We don't care enough to support this.

10. *Is section 2.14.8, "Color and Associated Data Clipping" in the core specification still correct?*

    DISCUSSION: This section is in need of some updating, now that varying variables can be interpolated without perspective correction. Some (not so precise) language has been added in the spec body, suggesting that the interpolation needs to be performed in such a way as to produce results that vary linearly in screen space. However, we could define the exact interpolation method required to achieve this. A suggested updated paragraph follows, but we'll leave updating section 2.14.8 to a future edit of the core specification, not this extension.

    Replace Section 2.14.8, and rename it to "Vertex Attribute Clipping"

    After lighting, clamping or masking and possible flatshading, vertex attributes, including colors, texture and fog coordinates, shader varying variables, and point sizes computed on a per vertex basis, are clipped. Those attributes associated with a vertex that lies within the clip volume are unaffected by clipping. If a primitive is clipped, however, the attributes assigned to vertices produced by clipping are produced by interpolating attributes along the clipped edge.

    Let the attributes assigned to the two vertices P_1 and P_2 of an unclipped edge be a_1 and a_2. The value of t (section 2.12) for a clipped point P is used to obtain the attribute associated with P as

    ```
    a = t * a_1 + (1-t) * a_2
    ```

    unless the attribute is specified to be interpolated without perspective correction in a shader (using the noperspective keyword). In that case, the attribute associated with P is

    ```
    a = t' * a_1 + (1-t') * a_2
    ```

    where

    ```
    t' = (t * w_1) / (t * w_1 + (1-t) * w_2)
    ```

    and w_1 and w_2 are the w clip coordinates of P_1 and P_2, respectively. If w_1 or w_2 is either zero or negative, the value of the associated attribute is undefined.

    For a color index color, multiplying a color by a scalar means multiplying the index by the scalar. For a vector attribute, it means multiplying each vector component by the scalar. Polygon clipping may create a clipped vertex along an edge of the clip volume's boundary. This situation is handled by noting that polygon clipping proceeds by clipping against one plane of the clip volume's boundary at a time. Attribute clipping is done in the same way, so that clipped points always occur at the intersection of polygon edges (possibly already clipped) with the clip volume's boundary.

11. *When and where in the texture filtering process are texel offsets applied?*

    DISCUSSION: Texel offsets are applied to the (u,v,w) coordinates of the base level of the texture if the texture filter mode does not indicate mipmapping. Otherwise, texel offsets are applied to the (u,v,w) coordinates of the mipmap level 'd', as found by equation 3.27 or to mipmap levels 'd1' and 'd2' as found by equation 3.28 in the OpenGL 2.0 specification. In other words, texel offsets are applied to the (u,v,w) coordinate of whatever mipmap level is accessed.

12. *Why is writing to the built-in output variable "gl_Position" in a vertex shader now optional?*

    DISCUSSION: Before this specification, writing to gl_Position in a vertex shader was mandatory. The GL pipeline required a vertex position to be written in order to produce well-defined output. This is still the case if the GL pipeline indeed needs a vertex position. However, with fourth-generation programmable hardware there are now cases where the GL pipeline no longer requires a vertex position in order to produce well-defined results. If a geometry shader is present, the vertex shader does not need to write to gl_Position anymore. Instead, the geometry shader can compute a vertex position and write to its gl_Position output. In case of transform-feedback, where the output of a vertex or geometry shader is streamed to one or more buffer objects, perfectly valid results can be obtained without either the vertex shader nor geometry shader writing to gl_Position. The transform-feedback specification adds a new enable to discard primitives right before rasterization, making it potentially unnecessary to write to gl_Position.

Revision History

Rev.	Date	Author	Changes
12	02/04/08	pbrown	Fix errors in texture wrap mode handling. Added a missing clamp to avoid sampling border in REPEAT mode. Fixed incorrectly specified weights for LINEAR filtering.
11	05/08/07	pbrown	Add VertexAttribIPointerEXT to the list of commands that can't go in display lists.
10	01/23/07	pbrown	Fix prototypes for a variety of functions that were specified with an incorrect sampler type.
9	12/15/06	pbrown	Documented that the '#extension' token for this extension should begin with "GL_", as apparently called for per convention.
8	--		Pre-release revisions.

Name

    EXT_multi_draw_arrays

Name Strings

    GL_EXT_multi_draw_arrays

Version

    $Date: 1998/04/03 04:35:50 $ $Revision: 1.1 $

Number

    148

Dependencies

    OpenGL 1.1 is required. The language is written against the OpenGL 1.2
    specification.

Overview

    These functions behave identically to the standard OpenGL 1.1 functions
    glDrawArrays() and glDrawElements() except they handle multiple lists of
    vertices in one call. Their main purpose is to allow one function call
    to render more than one primitive such as triangle strip, triangle fan,
    etc.

New Procedures and Functions

```
    void glMultiDrawArraysEXT( GLenum mode,
                               GLint *first,
                               GLsizei *count,
                               GLsizei primcount)
    Parameters
    ----------
        mode            Specifies what kind of primitives to
                        render. Symbolic constants GL_POINTS,
                        GL_LINE_STRIP, GL_LINE_LOOP, GL_LINES,
                        GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN,
                        GL_TRIANGLES, GL_QUAD_STRIP, GL_QUADS,
                        and GL_POLYGON are accepted.

        first           Points to an array of starting indices in
                        the enabled arrays.

        count           Points to an array of the number of indices
                        to be rendered.

        primcount       Specifies the size of first and count
    void glMultiDrawElementsEXT( GLenum mode,
                                 GLsizei *count,
                                 GLenum type,
                                 const GLvoid **indices,
                                 GLsizei primcount)

    Parameters
    ----------
        mode            Specifies what kind of primitives to render.
                        Symbolic constants GL_POINTS, GL_LINE_STRIP,
                        GL_LINE_LOOP, GL_LINES, GL_TRIANGLE_STRIP,
                        GL_TRIANGLE_FAN, GL_TRIANGLES, GL_QUAD_STRIP,
                        GL_QUADS, and GL_POLYGON are accepted.

        count           Points to and array of the element counts

        type            Specifies the type of the values in indices.
                        Must be  one  of GL_UNSIGNED_BYTE,
                        GL_UNSIGNED_SHORT, or GL_UNSIGNED_INT.

        indices         Specifies a  pointer to the location where
                        the indices are stored.

        primcount       Specifies the size of the count array
```

New Tokens

    None

Additions to Chapter 2 of the 1.2 Specification (OpenGL Operation)

    Section 2.8 Vertex Arrays:

    The command

```
        void glMultiDrawArraysEXT( GLenum mode,
                                   GLint* first,
                                   GLsizei *count,
                                   GLsizei primcount)
```

    Behaves identically to DrawArrays except that a list of arrays is
    specified instead. The number of lists is specified in the primcount
    parameter. It has the same effect as:

```
        for(i=0; i<primcount; i++) {
           if (*(count+i)>0) DrawArrays(mode, *(first+i), *(count+i));
        }
```

The command

```
    void glMultiDrawElementsEXT( GLenum mode,
                                 GLsizei *count,
                                 GLenum type,
                                 const GLvoid **indices,
                                 GLsizei primcount)
```

Behaves identically to DrawElements except that a list of arrays is specified instead. The number of lists is specified in the primcount parameter. It has the same effect as:

```
    for(i=0; i<primcount; i++) {
        if (*(count+i)>0) DrawElements(mode, *(count+i), type,
                                       *(indices+i));
    }
```

Additions to Chapter 3 of the 1.2 Specification (Rasterization)

None.

Additions to Chapter 4 of the 1.2 Specification (Per-Fragment Operations and

None.

Additions to Chapter 5 of the 1.2 Specification (Special Functions)

None.

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

None.

Additions to the GLX Specification

None.

GLX Protocol

None.

Errors

GL_INVALID_ENUM is generated if <mode> is not an accepted value.

GL_VALUE is generated if <primcount> is negative.

GL_INVALID_OPERATION is generated if glMultiDrawArraysEXT or glMultiDrawElementsEXT is executed between the execution of glBegin and the corresponding glEnd.

New State

None.

Name

    EXT_packed_float

Name Strings

    GL_EXT_packed_float
    WGL_EXT_pixel_format_packed_float
    GLX_EXT_fbconfig_packed_float

Contact

    Mark J. Kilgard, NVIDIA Corporation (mjk 'at' nvidia.com)

Status

    Shipping for GeForce 8 Series (November 2006)

Version

    Date: November 6, 2006
    Revision: 0.4

Number

    328

Dependencies

    OpenGL 1.1 required

    ARB_color_buffer_float affects this extension.

    EXT_texture_shared_exponent trivially affects this extension.

    EXT_framebuffer_object affects this extension.

    WGL_ARB_pixel_format is required for use with WGL.

    WGL_ARB_pbuffer affects WGL pbuffer support for this extension.

    GLX 1.3 is required for use with GLX.

    This extension is written against the OpenGL 2.0 (September 7,
    2004) specification.

Overview

    This extension adds a new 3-component floating-point texture format
    that fits within a single 32-bit word. This format stores 5 bits
    of biased exponent per component in the same manner as 16-bit
    floating-point formats, but rather than 10 mantissa bits, the red,
    green, and blue components have 6, 6, and 5 bits respectively.
    Each mantissa is assumed to have an implied leading one except in the
    denorm exponent case. There is no sign bit so only non-negative
    values can be represented. Positive infinity, positive denorms,

    and positive NaN values are representable. The value of the fourth
    component returned by a texture fetch is always 1.0.

    This extension also provides support for rendering into an unsigned
    floating-point rendering format with the assumption that the texture
    format described above could also be advertised as an unsigned
    floating-point format for rendering.

    The extension also provides a pixel external format for specifying
    packed float values directly.

New Procedures and Functions

    None

New Tokens

```
    Accepted by the <internalformat> parameter of TexImage1D,
    TexImage2D, TexImage3D, CopyTexImage1D, CopyTexImage2D, and
    RenderbufferStorageEXT:

        R11F_G11F_B10F_EXT                          0x8C3A

    Accepted by the <type> parameter of DrawPixels, ReadPixels,
    TexImage1D, TexImage2D, GetTexImage, TexImage3D, TexSubImage1D,
    TexSubImage2D, TexSubImage3D, GetHistogram, GetMinmax,
    ConvolutionFilter1D, ConvolutionFilter2D, ConvolutionFilter3D,
    GetConvolutionFilter, SeparableFilter2D, GetSeparableFilter,
    ColorTable, ColorSubTable, and GetColorTable:

        UNSIGNED_INT_10F_11F_11F_REV_EXT            0x8C3B

    Accepted by the <pname> parameters of GetIntegerv, GetFloatv, and
    GetDoublev:

        RGBA_SIGNED_COMPONENTS_EXT                  0x8C3C

    Accepted as a value in the <piAttribIList> and <pfAttribFList>
    parameter arrays of wglChoosePixelFormatARB, and returned in the
    <piValues> parameter array of wglGetPixelFormatAttribivARB, and the
    <pfValues> parameter array of wglGetPixelFormatAttribfvARB:

        WGL_TYPE_RGBA_UNSIGNED_FLOAT_EXT            0x20A8

    Accepted as values of the <render_type> arguments in the
    glXCreateNewContext and glXCreateContext functions

        GLX_RGBA_UNSIGNED_FLOAT_TYPE_EXT            0x20B1

    Returned by glXGetFBConfigAttrib (when <attribute> is set to
    GLX_RENDER_TYPE) and accepted by the <attrib_list> parameter of
    glXChooseFBConfig (following the GLX_RENDER_TYPE token):

        GLX_RGBA_UNSIGNED_FLOAT_BIT_EXT              0x00000008
```

Additions to Chapter 2 of the 2.0 Specification (OpenGL Operation)

-- Add two new sections after Section 2.1.2, (page 6):

2.1.A Unsigned 11-Bit Floating-Point Numbers

An unsigned 11-bit floating-point number has no sign bit, a 5-bit exponent (E), and a 6-bit mantissa (M). The value of an unsigned 11-bit floating-point number (represented as an 11-bit unsigned integer N) is determined by the following:

```
    0.0,                        if E == 0 and M == 0,
    2^-14 * (M / 64),           if E == 0 and M != 0,
    2^(E-15) * (1 + M/64),      if 0 < E < 31,
    INF,                        if E == 31 and M == 0, or
    NaN,                        if E == 31 and M != 0,
```

where

```
    E = floor(N / 64), and
    M = N mod 64.
```

Implementations are also allowed to use any of the following alternative encodings:

```
    0.0,                        if E == 0 and M != 0
    2^(E-15) * (1 + M/64)       if E == 31 and M == 0
    2^(E-15) * (1 + M/64)       if E == 31 and M != 0
```

When a floating-point value is converted to an unsigned 11-bit floating-point representation, finite values are rounded to the closet representable finite value. While less accurate, implementations are allowed to always round in the direction of zero. This means negative values are converted to zero. Likewise, finite positive values greater than 65024 (the maximum finite representable unsigned 11-bit floating-point value) are converted to 65024. Additionally: negative infinity is converted to zero; positive infinity is converted to positive infinity; and both positive and negative NaN are converted to positive NaN.

Any representable unsigned 11-bit floating-point value is legal as input to a GL command that accepts 11-bit floating-point data. The result of providing a value that is not a floating-point number (such as infinity or NaN) to such a command is unspecified, but must not lead to GL interruption or termination. Providing a denormalized number or negative zero to GL must yield predictable results.

2.1.B Unsigned 10-Bit Floating-Point Numbers

An unsigned 10-bit floating-point number has no sign bit, a 5-bit exponent (E), and a 5-bit mantissa (M). The value of an unsigned 10-bit floating-point number (represented as an 10-bit unsigned integer N) is determined by the following:

```
    0.0,                      if E == 0 and M == 0,
    2^-14 * (M / 32),         if E == 0 and M != 0,
    2^(E-15) * (1 + M/32),    if 0 < E < 31,
    INF,                      if E == 31 and M == 0, or
    NaN,                      if E == 31 and M != 0,
```

where

```
    E = floor(N / 32), and
    M = N mod 32.
```

When a floating-point value is converted to an unsigned 10-bit floating-point representation, finite values are rounded to the closet representable finite value. While less accurate, implementations are allowed to always round in the direction of zero. This means negative values are converted to zero. Likewise, finite positive values greater than 64512 (the maximum finite representable unsigned 10-bit floating-point value) are converted to 64512. Additionally: negative infinity is converted to zero; positive infinity is converted to positive infinity; and both positive and negative NaN are converted to positive NaN.

Any representable unsigned 10-bit floating-point value is legal as input to a GL command that accepts 10-bit floating-point data. The result of providing a value that is not a floating-point number (such as infinity or NaN) to such a command is unspecified, but must not lead to GL interruption or termination. Providing a denormalized number or negative zero to GL must yield predictable results.

Additions to Chapter 3 of the 2.0 Specification (Rasterization)

-- Section 3.6.4, Rasterization of Pixel Rectangles

Add a new row to Table 3.5 (page 128):

type Parameter Token Name	Corresponding GL Data Type	Special Interpretation
UNSIGNED_INT_10F_11F_11F_REV_EXT	uint	yes

Add a new row to table 3.8: Packed pixel formats (page 132):

type Parameter Token Name	GL Data Type	Number of Components	Matching Pixel Formats
UNSIGNED_INT_10F_11F_11F_REV_EXT	uint	3	RGB

    Add a new entry to table 3.11: UNSIGNED_INT formats (page 134):

        UNSIGNED_INT_10F_11F_11F_REV_EXT:

```
  31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
  +----------------------------+-------------------------------+---------------------------------+
  |          3rd               |            2nd                |             1st                 |
  +----------------------------+-------------------------------+---------------------------------+
```

    Add to the end of the 2nd paragraph starting "Pixels are draw using":

    "If type is UNSIGNED_INT_10F_11F_11F_REV_EXT and format is not RGB
    then the error INVALID_ENUM occurs."

    Add UNSIGNED_INT_10F_11F_11F_REV_EXT to the list of packed formats
    in the 10th paragraph after the "Packing" subsection (page 130).

    Add before the 3rd paragraph (page 135, starting "Calling DrawPixels
    with a type of BITMAP...") from the end of the "Packing" subsection:

    "Calling DrawPixels with a type of UNSIGNED_INT_10F_11F_11F_REV_EXT
    and format of RGB is a special case in which the data are a series
    of GL uint values. Each uint value specifies 3 packed components
    as shown in table 3.11. The 1st, 2nd, and 3rd components are
    called f_red (11 bits), f_green (11 bits), and f_blue (10 bits)
    respectively.

    f_red and f_green are treated as unsigned 11-bit floating-point values
    and converted to floating-point red and green components respectively
    as described in section 2.1.A. f_blue is treated as an unsigned
    10-bit floating-point value and converted to a floating-point blue
    component as described in section 2.1.B."

-- Section 3.8.1, Texture Image Specification:

    "Alternatively if the internalformat is R11F_G11F_B10F_EXT, the red,
    green, and blue bits are converted to unsigned 11-bit, unsigned
    11-bit, and unsigned 10-bit floating-point values as described
    in sections 2.1.A and 2.1.B. These encoded values can be later
    decoded back to floating-point values due to texture image sampling
    or querying."

    Add a new row to Table 3.16 (page 154).

Sized Internal Format	Base Internal Format	R bits	G bits	B bits	A bits	L bits	I bits	D bits
R11F_G11F_B10F_EXT	RGB	11	11	10				

Additions to Chapter 4 of the 2.0 Specification (Per-Fragment Operations and the Frame Buffer)

-- Modify Chapter 4 Introduction, (page 198)

Modify first sentence of third paragraph (page 198):

"Color buffers consist of either signed or unsigned integer color indices, R, G, B and optionally A signed or unsigned integer values, or R, G, B, and optionally A signed or unsigned floating-point values."

-- Section 4.3.2, Reading Pixels

Add a row to table 4.7 (page 224);

type Parameter	GL Data Type	Component Conversion Formula
UNSIGNED_INT_10F_11F_11F_REV_EXT	uint	special

Replace second paragraph of "Final Conversion" (page 222) to read:

For an RGBA color, if <type> is not one of FLOAT, UNSIGNED_INT_5_9_9_9_REV_EXT, or UNSIGNED_INT_10F_11F_11F_REV_EXT, or if the CLAMP_READ_COLOR_ARB is TRUE, or CLAMP_READ_COLOR_ARB is FIXED_ONLY_ARB and the selected color (or texture) buffer is a fixed-point buffer, each component is first clamped to [0,1]. Then the appropriate conversion formula from table 4.7 is applied the component."

Add a paragraph after the second paragraph of "Final Conversion" (page 222):

"In the special case when calling ReadPixels with a type of UNSIGNED_INT_10F_11F_11F_REV_EXT and format of RGB, the conversion is done as follows: The returned data are packed into a series of GL uint values. The red, green, and blue components are converted to unsigned 11-bit floating-point, unsigned 11-bit floating-point, and unsigned 10-bit floating point as described in section 2.1.A and 2.1.B. The resulting red 11 bits, green 11 bits, and blue 10 bits are then packed as the 1st, 2nd, and 3rd components of the UNSIGNED_INT_10F_11F_11F_REV_EXT format as shown in table 3.11."

Additions to Chapter 5 of the 2.0 Specification (Special Functions)

None

Additions to Chapter 6 of the 2.0 Specification (State and State Requests)

None

Additions to the OpenGL Shading Language specification

None

Additions to Chapter 3 of the GLX 1.3 Specification (Functions and Errors)

Replace Section 3.3.3 (p.12) Paragraph 4 to:

The attribute GLX_RENDER_TYPE has as its value a mask indicating what type of GLXContext a drawable created with the corresponding GLXFBConfig can be bound to. The following bit settings are supported: GLX_RGBA_BIT, GLX_RGBA_FLOAT_BIT, GLX_RGBA_UNSIGNED_FLOAT_BIT, GLX_COLOR_INDEX_BIT. If combinations of bits are set in the mask then drawables created with the GLXFBConfig can be bound to those corresponding types of rendering contexts.

Add to Section 3.3.3 (p.15) after first paragraph:

Note that unsigned floating point rendering is only supported for GLXPbuffer drawables. The GLX_DRAWABLE_TYPE attribute of the GLXFBConfig must have the GLX_PBUFFER_BIT bit set and the GLX_RENDER_TYPE attribute must have the GLX_RGBA_UNSIGNED_FLOAT_BIT set. Unsigned floating point rendering assumes the framebuffer format has no sign bits so all component values are non-negative. In contrast, conventional floating point rendering assumes signed components.

Modify Section 3.3.7 (p.25 Rendering Contexts) remove period at end of second paragraph and replace with:

; if render_type is set to GLX_RGBA_UNSIGNED_FLOAT_TYPE then a context that supports unsigned floating point RGBA rendering is created.

GLX Protocol

None.

Additions to the WGL Specification

Modify the values accepted by WGL_PIXEL_TYPE_ARB to:

WGL_PIXEL_TYPE_ARB
The type of pixel data. This can be set to WGL_TYPE_RGBA_ARB, WGL_TYPE_RGBA_FLOAT_ARB, WGL_TYPE_RGBA_UNSIGNED_FLOAT_EXT, or WGL_TYPE_COLORINDEX_ARB.

Add this explanation of unsigned floating point rendering:

"Unsigned floating point rendering assumes the framebuffer format has no sign bits so all component values are non-negative. In contrast, conventional floating point rendering assumes signed components."

Dependencies on WGL_ARB_pbuffer

Ignore the "Additions to the WGL Specification" section if WGL_ARB_pbuffer is not supported.

Dependencies on WGL_ARB_pixel_format

    The WGL_ARB_pixel_format extension must be used to determine a
    pixel format with unsigned float components.

Dependencies on ARB_color_buffer_float

    If ARB_color_buffer_float is not supported, replace this amended
    sentence from 4.3.2 above

    For an RGBA color, if <type> is not one of FLOAT,
    UNSIGNED_INT_5_9_9_9_REV_EXT, or UNSIGNED_INT_10F_11F_11F_REV_EXT,
    or if the CLAMP_READ_COLOR_ARB is TRUE, or CLAMP_READ_COLOR_ARB
    is FIXED_ONLY_ARB and the selected color (or texture) buffer is
    a fixed-point buffer, each component is first clamped to [0,1]."

    with

    "For an RGBA color, if <type> is not one of FLOAT,
    UNSIGNED_INT_5_9_9_9_REV_EXT, or UNSIGNED_INT_10F_11F_11F_REV_EXT
    and the selected color buffer (or texture image for GetTexImage)
    is a fixed-point buffer (or texture image for GetTexImage), each
    component is first clamped to [0,1]."

Dependencies on EXT_texture_shared_exponent

    If EXT_texture_shared_exponent is not supported, delete the reference
    to UNSIGNED_INT_5_9_9_9_REV_EXT in section 4.3.2.

Dependencies on EXT_framebuffer_object

    If EXT_framebuffer_object is not supported, then
    RenderbufferStorageEXT is not supported and the R11F_G11F_B10F_EXT
    internalformat is therefore not supported by RenderbufferStorageEXT.

    If EXT_framebuffer_object is supported, glRenderbufferStorageEXT
    accepts GL_RG11F_B10F_EXT for its internalformat parameter because
    GL_RG11F_B10F_EXT has a base internal format of GL_RGB that is listed
    as color-renderable by the EXT_framebuffer_object specification.

Errors

    Relaxation of INVALID_ENUM errors

    TexImage1D, TexImage2D, TexImage3D, CopyTexImage1D, CopyTexImage2D,
    and RenderbufferStorageEXT accept the new R11F_G11F_B10F_EXT token
    for internalformat.

    DrawPixels, ReadPixels, TexImage1D, TexImage2D, GetTexImage,
    TexImage3D, TexSubImage1D, TexSubImage2D, TexSubImage3D,
    GetHistogram, GetMinmax, ConvolutionFilter1D, ConvolutionFilter2D,
    ConvolutionFilter3D, GetConvolutionFilter, SeparableFilter2D,
    GetSeparableFilter, ColorTable, ColorSubTable, and GetColorTable
    accept the new UNSIGNED_INT_10F_11F_11F_REV_EXT token for type.

New errors

INVALID_OPERATION is generated by DrawPixels, ReadPixels, TexImage1D, TexImage2D, GetTexImage, TexImage3D, TexSubImage1D, TexSubImage2D, TexSubImage3D, GetHistogram, GetMinmax, ConvolutionFilter1D, ConvolutionFilter2D, ConvolutionFilter3D, GetConvolutionFilter, SeparableFilter2D, GetSeparableFilter, ColorTable, ColorSubTable, and GetColorTable if <type> is UNSIGNED_INT_10F_11F_11F_REV_EXT and <format> is not RGB.

New State

In table 6.17, Textures (page 278), increment the 42 in "n x Z42*" by 1 for the R11F_G11F_B10F_EXT format.

[NOTE: The OpenGL 2.0 specification actually should read "n x Z48*" because of the 6 generic compressed internal formats in table 3.18.]

(modify table 6.33, p. 294)

Get Value	Type	Get Command	Initial Value	Description	Sec.	Attribute
RGBA_SIGNED_COMPONENTS_EXT	4xB	GetIntegerv	-	True if respective R, G, B, and A components are signed	4	-

New Implementation Dependent State

None

Issues

1) What should this extension be called?

RESOLVED: EXT_packed_float

This extension provides a new 3-component packed float format for use as a texture internal format, pixel external format, and framebuffer color format.

"packed" indicates the extension is packing components at reduced precisions (similar to EXT_packed_pixels or NV_packed_depth_stencil).

EXT_r11f_g11f_b10f_float was considered but there's no precedent for extension names to be so explicit (or cryptic?) about format specifics in the extension name.

2) Should there be an rgb11f_b10f framebuffer format?

RESOLVED: Yes. Unsigned floating-point rendering formats for GLX and WGL are provided. The assumption is that this functionality could be used to advertise a pixel format with 11 bits of unsigned floating-point red, 11 bits of unsigned floating-point green, and 10 bits of floating-point blue.

In theory, an implementation could advertise other component sizes other than 11/11/10 for an unsigned floating-point framebuffer format but that is not expected.

3) *Should there be GLX and WGL extension strings?*

RESOLVED: Yes, there are WGL and GLX tokens added to support querying unsigned floating-point color buffer formats named WGL_EXT_pixel_format_packed_float and GLX_EXT_fbconfig_packed_float respectively.

4) *Should there be an unequal distribution of red, green, and blue mantissa bits?*

RESOLVED: Yes. A 6-bit mantissa for red and green is unbalanced with the 5-bit mantissa for blue, but this allows all the bits of a 32 bit word (6+6+5+3*5=32) to be used. The blue component is chosen to have fewer bits because 1) it is the third component, and 2) there's a belief that the human eye is less sensitive to blue variations..

Developers should be aware that subtle yellowing or bluing of gray-scale values is possible because of the extra bit of mantissa in the red and green components.

5) *Should there be an external format for r11f_g11f_b10f?*

RESOLVED: Yes. This makes it fast to load GL_R11F_G11F_B10F_EXT textures without any translation by the driver.

6) *What is the exponent bias?*

RESOLVED: 15, just like 16-bit half-precision floating-point values.

7) *Can s10e5 floating-point filtering be used to filter r11f_g11f_b10f values? If so, how?*

RESOLVED: Yes. It is easy to promote r11f_g11f_b10f values to s10e5 components.

8) *Should automatic mipmap generation be supported for r11f_g11f_b10f textures?*

RESOLVED: Yes.

9) *Should non-texture and non-framebuffer commands for loading pixel data accept the GL_UNSIGNED_INT_10F_11F_11F_REV_EXT type?*

RESOLVED: Yes.

Once the pixel path has to support the new type/format combination of GL_UNSIGNED_INT_5_9_9_9_REV_EXT / GL_RGB for specifying and querying texture images, it might as well be supported for all commands that pack and unpack RGB pixel data.

The specification is written such that the glDrawPixels type/format parameters are accepted by glReadPixels, glTexGetImage, glTexImage2D, and other commands that are specified in terms of glDrawPixels.

10) *Should non-texture internal formats (such as for color tables, convolution kernels, histogram bins, and min/max tables) accept GL_R11F_G11F_B10F_EXT format?*

RESOLVED: No.

That's pointless. No hardware is ever likely to support GL_R11F_G11F_B10F_EXT internal formats for anything other than textures and maybe color buffers in the future. This format is not interesting for color tables, convolution kernels, etc.

11) *Should a format be supported with sign bits for each component?*

RESOLVED: No. A sign bit for each of the three components would steal too many bits from the mantissa. This format is intended for storing radiance and irradiance values that are physically non-negative.

12) *Should we support a non-REV version of the GL_UNSIGNED_INT_10F_11F_11F_REV_EXT token?*

RESOLVED: No. We don't want to promote different arrangements of the bitfields for r11f_g11f_b10f values.

13) *Can you use the GL_UNSIGNED_INT_10F_11F_11F_REV_EXT format with just any format?*

RESOLVED: You can only use the GL_UNSIGNED_INT_10F_11F_11F_REV_EXT format with GL_RGB. Otherwise, the GL generates an GL_INVALID_OPERATION error. Just as the GL_UNSIGNED_BYTE_3_3_2 format just works with GL_RGB (or else the GL generates an GL_INVALID_OPERATION error), so should GL_UNSIGNED_INT_10F_11F_11F_REV_EXT.

14) *Should blending be supported for a packed float framebuffer format?*

RESOLVED: Yes. Blending is required for other floating-point framebuffer formats introduced by ARB_color_buffer_float. The equations for blending should be evaluated with signed floating-point math but the result will have to be clamped to non-negative values to be stored back into the packed float format of the color buffer.

15) *Should unsigned floating-point framebuffers be queried differently from conventional (signed) floating-point framebuffers?*

RESOLVED: Yes. An existing application using ARB_color_buffer_float can rightfully expect a floating-point color buffer format to provide signed components. The packed float format does not provide a sign bit. Simply treating packed float color buffer formats as floating-point might break some existing applications that depend on a float color buffer to be signed.

For this reason, there are new WGL_TYPE_RGBA_UNSIGNED_FLOAT_EXT (for WGL) and GLX_RGBA_UNSIGNED_FLOAT_BIT_EXT (for GLX) framebuffer format parameters.

16) *What should glGet of GL_RGBA_FLOAT_MODE_ARB return for unsigned float color buffer formats?*

RESOLVED. GL_RGBA_FLOAT_MODE_ARB should return true. The packed float components are unsigned but still floating-point.

17) *Can you query with glGet to determine if the color buffer has unsigned float components?*

RESOLVED: Yes. Call glGetIntegerv on GL_RGBA_SIGNED_COMPONENTS_EXT. The value returned is a 4-element array. Element 0 corresponds to red, element 1 corresponds to green, element 2 corresponds to blue, and element 3 corresponds to alpha. If a color component is signed, its corresponding element is true (GL_TRUE). This is the same way the GL_COLOR_WRITEMASK bits are formatted.

For the packed float format, all the elements are zeroed since the red, green, and blue components are unsigned and the alpha component is non-existent. All elements are also zeroed for conventional fixed-point color buffer formats. Elements are set for signed floating-point formats such as those introduced by ARB_color_buffer_float. If a component (such as alpha) has zero bits, the component should not be considered signed and so the bit for the respective component should be zeroed.

This generality allows a future extension to specify float color buffer formats that had a mixture of signed and unsigned floating-point components. However, this extension only provides a packed float color format with all unsigned components.

18) *How many bits of alpha should GL_ALPHA_BITS return for the packed float color buffer format?*

RESOLVED: Zero.

19) *Can you render to a packed float texture with the EXT_framebuffer_object functionality?*

RESOLVED: Yes.

Potentially an implementation could return GL_FRAMEBUFFER_UNSUPPORTED_EXT when glCheckFramebufferStatusEXT for a framebuffer object including a packed float color buffer, but implementations are likely to support (and strongly encouraged to support) the packed float format for use with a framebuffer object because the packed float format is expected to be a memory-efficient floating-point color format well-suited for rendering, particularly rendering involving high-dynamic range.

20) *This extension is for a particular packed float format. What if new packed float formats come along?*

RESOLVED: A new extension could be introduced with a name like EXT_packed_float2, but at this time, no other such extensions are expected except for the EXT_texture_shared_exponent extension. It simply hard to justify packing three or more components into a single 32-bit word in lots of different ways since any approach is going to be a compromise of some sort. For two-component or one-component floating-point formats, the existing ARB_texture_float formats fit nicely into 16 or 32 bits by simply using half precision floating-point. If 64 bits are allowed for a pixel, the GL_RGBA16F_ARB is a good choice.

The packed float format is similar to the format introduced by the EXT_texture_shared_exponent extension, but that extension is not a pure packed float format. Unlike the packed float format, the EXT_texture_shared_exponent format shares a single exponent between the RGB components rather than providing an independent exponent for each component. Because the EXT_texture_shared_exponent uses fewer bits to store exponent values, more mantissa precision is provided.

21) *Should this extension provide pbuffer support?*

RESOLVED: Yes. Pbuffers are core GLX 1.3 functionality. While using FBO is probably the preferred way to render to r11f_g11f_b10f framebuffers but pbuffer support is natural to provide. WGL should have r11f_g11f_b10f pbuffer support too.

22) *Must an implementation support NaN, Infinity, and/or denorms?*

RESOLVED: The preferred implementation is to support NaN, Infinity, and denorms. Implementations are allowed to flush denorms to zero, and treat NaN and Infinity values as large finite values.

This allowance flushes denorms to zero:

```
    0.0,                      if E == 0 and M != 0
```

This allowance treats Infinity as a finite value:

```
    2^16                      if E == 31 and M == 0
```

This allowance treats NaN encodings as finite values:

```
    2^16 * (1 + M/64)         if E == 31 and M != 0
```

The expectation is that mainstream GPUs will support NaN, Infinity, and denorms while low-end implementations such as for OpenGL ES 2.0 will likely support denorms but neither NaN nor Infinity.

    There is not an indication of how these floating-point special
    values are treated (though an application could test an
    implementation if necessary).

23) *Should this extension interoperate with framebuffer objects?*

    RESOLVED: Definitely. No particular specification language is
    required.

    In particular, glRenderbufferStorageEXT should accept
    GL_R11F_G11F_B10F_EXT for its internalformat parameter (true
    because this extension adds a new format to Table 3.16).

24) *Are negative color components clamped to zero when written into
    an unsigned floating-point color buffer? If so, do we need to
    say in the Blending or Dithering language that negative color
    components are clamped to zero?*

    RESOLVED: Yes, negative color components are clamped to
    zero when written to an unsigned floating-point color buffer.
    No specification language is required for this behavior because
    the ARB_color_buffer_float extension says

    "In RGBA mode dithering selects, for each color component, either
    the most positive representable color value (for that particular
    color component) that is less than or equal to the incoming
    color component value, c, or the most negative representable
    color value that is greater than or equal to c.

    If dithering is disabled, then each incoming color component
    c is replaced with the most positive representable color value
    (for that particular component) that is less than or equal to c,
    or by the most negative representable value, if no representable
    value is less than or equal to c;"

    The most negative representable value for unsigned
    floating-point values is zero. So the existing language from
    ARB_color_buffer_float already indicates that negative values
    are clamped to zero for unsigned floating-point color buffers.
    No additional specification language is required.

25) *Prior texture internal formats have generic formats (example:
    GL_RGB) and corresponding sized formats (GL_RGB8, GL_RGB10,
    etc.). Should we add a generic format corresponding to
    GL_R11F_G11F_B10F_EXT?*

    RESOLVED: No. It's unlikely there will be any other unsigned
    floating-point texture formats.

Revision History

None

Name

    EXT_packed_pixels

Name Strings

    GL_EXT_packed_pixels

Version

    $Date: 1997/09/22 23:23:58 $ $Revision: 1.21 $

Number

    23

Dependencies

```
EXT_abgr affects the definition of this extension
EXT_texture3D affects the definition of this extension
EXT_subtexture affects the definition of this extension
EXT_histogram affects the definition of this extension
EXT_convolution affects the definition of this extension
SGI_color_table affects the definition of this extension
SGIS_texture4D affects the definition of this extension
EXT_cmyka affects the definition of this extension
```

Overview

```
This extension provides support for packed pixels in host memory.  A
packed pixel is represented entirely by one unsigned byte, one
unsigned short, or one unsigned integer.  The fields with the packed
pixel are not proper machine types, but the pixel as a whole is.  Thus
the pixel storage modes, including PACK_SKIP_PIXELS, PACK_ROW_LENGTH,
PACK_SKIP_ROWS, PACK_IMAGE_HEIGHT_EXT, PACK_SKIP_IMAGES_EXT,
PACK_SWAP_BYTES, PACK_ALIGNMENT, and their unpacking counterparts all
work correctly with packed pixels.
```

New Procedures and Functions

    None

New Tokens

```
Accepted by the <type> parameter of DrawPixels, ReadPixels, TexImage1D,
TexImage2D, GetTexImage, TexImage3DEXT, TexSubImage1DEXT,
TexSubImage2DEXT, TexSubImage3DEXT, GetHistogramEXT, GetMinmaxEXT,
ConvolutionFilter1DEXT, ConvolutionFilter2DEXT, ConvolutionFilter3DEXT,
GetConvolutionFilterEXT, SeparableFilter2DEXT, SeparableFilter3DEXT,
GetSeparableFilterEXT, ColorTableSGI, GetColorTableSGI, TexImage4DSGIS,
and TexSubImage4DSGIS:

    UNSIGNED_BYTE_3_3_2_EXT         0x8032
    UNSIGNED_SHORT_4_4_4_4_EXT      0x8033
    UNSIGNED_SHORT_5_5_5_1_EXT      0x8034
    UNSIGNED_INT_8_8_8_8_EXT        0x8035
    UNSIGNED_INT_10_10_10_2_EXT     0x8036
```

Additions to Chapter 2 of the 1.0 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.0 Specification (Rasterization)

The five tokens defined by this extension are added to Table 3.4:

<type> Parameter Token Value	Corresponding GL Data Type	Special Interpretation
UNSIGNED_BYTE	ubyte	No
BYTE	byte	No
UNSIGNED_SHORT	ushort	No
SHORT	short	No
UNSIGNED_INT	uint	No
INT	int	No
FLOAT	float	No
BITMAP	ubyte	Yes
UNSIGNED_BYTE_3_3_2_EXT	ubyte	Yes
UNSIGNED_SHORT_4_4_4_4_EXT	ushort	Yes
UNSIGNED_SHORT_5_5_5_1_EXT	ushort	Yes
UNSIGNED_INT_8_8_8_8_EXT	uint	Yes
UNSIGNED_INT_10_10_10_2_EXT	uint	Yes

Table 3.4: DrawPixels and ReadPixels <type> parameter values and the corresponding GL data types. Refer to table 2.2 for definitions of GL data types. Special interpretations are described near the end of section 3.6.3.

[Section 3.6.3 of the GL Specification (Rasterization of Pixel Rectangles) is rewritten as follows:]

3.6.3 Rasterization of Pixel Rectangles

The process of drawing pixels encoded in host memory is diagrammed in Figure 3.7. We describe the stages of this process in the order in which they occur.

Pixels are drawn using

```
    void DrawPixels(sizei width,
                    sizei height,
                    enum format,
                    enum type,
                    void* data);
```

<format> is a symbolic constant indicating what the values in memory represent. <width> and <height> are the width and height, respectively, of the pixel rectangle to be drawn. <data> is a pointer to the data to be drawn. These data are represented with one of seven GL data types, specified by <type>. The correspondence between the thirteen <type> token values and the GL data types they indicate is given in Table 3.4. If the GL is in color index mode and <format> is not one of COLOR_INDEX, STENCIL_INDEX, or DEPTH_COMPONENT, then the error INVALID_OPERATION

occurs. Some additional constraints on the combinations of <format> and <type> values that are accepted are discussed below.

Unpacking

Data are taken from host memory as a sequence of signed or unsigned bytes (GL data types byte and ubyte), signed or unsigned short integers (GL data types short and ushort), signed or unsigned integers (GL data types int and uint), or floating-point values (GL data type float). These elements are grouped into sets of one, two, three, four, or five values, depending on the <format>, to form a group. Table 3.5 summarizes the format of groups obtained from memory. It also indicates those formats that yield indices and those that yield components.

Format Name	Target Buffer	Element Meaning and Order
COLOR_INDEX	Color	Color index
STENCIL_INDEX	Stencil	Stencil index
DEPTH_COMPONENT	Depth	Depth component
RED	Color	R component
GREEN	Color	G component
BLUE	Color	B component
ALPHA	Color	A component
RGB	Color	R, G, B components
RGBA	Color	R, G, B, A components
ABGR_EXT	Color	A, B, G, R components
CMYK_EXT	Color	Cyan, Magenta, Yellow, Black components
CMYKA_EXT	Color	Cyan, Magenta, Yellow, Black, A components
LUMINANCE	Color	Luminance component
LUMINANCE_ALPHA	Color	Luminance, A components

Table 3.5: DrawPixels and ReadPixels formats. The third column gives a description of and the number and order of elements in a group.

By default the values of each GL data type are interpreted as they would be specified in the language of the client's GL binding. If UNPACK_SWAP_BYTES is set to TRUE, however, then the values are interpreted with the bit orderings modified as per the table below. The modified bit orderings are defined only if the GL data type ubyte has eight bits, and then for each specific GL data type only if that type is represented with 8, 16, or 32 bits.

Element Size	Default Bit Ordering	Modified Bit Ordering
8-bit	[7..0]	[7..0]
16-bit	[15..0]	[7..0] [15..8]
32-bit	[31..0]	[7..0] [15..8] [23..16] [31..24]

Table: Bit ordering modification of elements when UNPACK_SWAP_BYTES is TRUE. These reorderings are defined only when GL data type ubyte has 8 bits, and then only for GL data types with 8, 16, or 32 bits.

The groups in memory are treated as being arranged in a rectangle. This rectangle consists of a series of rows, with the first element of the first group of the first row pointed to by the pointer passed to DrawPixels. If the value of UNPACK_ROW_LENGTH is not positive, then the number of groups in a row is <width>; otherwise the number of groups is UNPACK_ROW_LENGTH. If the first element of the first row is at location p in memory, then the location of the first element of the Nth row is

```
    p + Nk
```

where N is the row number (counting from zero) and k is defined as

```
         / nl                        s >= a
    k = <
         \ a/s * ceiling(snl/a)      s < a
```

where n is the number of elements in a group, l is the number of groups in a row, a is the value of UNPACK_ALIGNMENT, and s is the size, in units of GL ubytes, of an element. If the number of bits per element is not 1, 2, 4, or 8 times the number of bits in a GL ubyte, then k = nl for all values of a.

There is a mechanism for selecting a sub-rectangle of groups from a larger containing rectangle. This mechanism relies on three integer parameters: UNPACK_ROW_LENGTH, UNPACK_SKIP_ROWS, and UNPACK_SKIP_PIXELS. Before obtaining the first group from memory, the pointer supplied to DrawPixels is effectively advanced by

```
    UNPACK_SKIP_PIXELS * n + UNPACK_SKIP_ROWS * k
```

elements. Then <width> groups are obtained from contiguous elements in memory (without advancing the pointer), after which the pointer is advanced by k elements. <height> sets of <width> groups of values are obtained this way. See Figure 3.8.

Calling DrawPixels with a <type> of UNSIGNED_BYTE_3_3_2, UNSIGNED_SHORT_4_4_4_4, UNSIGNED_SHORT_5_5_5_1, UNSIGNED_INT_8_8_8_8, or UNSIGNED_INT_10_10_10_2 is a special case in which all the elements of each group are packed into a single unsigned byte, unsigned short, or unsigned int, depending on the type. The number of elements per packed pixel is fixed by the type, and must match the number of elements per group indicated by the <format> parameter. (See the table below.) The error INVALID_OPERATION is generated if a mismatch occurs.

<type> Parameter Token Name	GL Data Type	Number of Elements	Matching Pixel Formats
UNSIGNED_BYTE_3_3_2_EXT	ubyte	3	RGB
UNSIGNED_SHORT_4_4_4_4_EXT	ushort	4	RGBA,ABGR_EXT,CMYK_EXT
UNSIGNED_SHORT_5_5_5_1_EXT	ushort	4	RGBA,ABGR_EXT,CMYK_EXT
UNSIGNED_INT_8_8_8_8_EXT	uint	4	RGBA,ABGR_EXT,CMYK_EXT
UNSIGNED_INT_10_10_10_2_EXT	uint	4	RGBA,ABGR_EXT,CMYK_EXT

Bitfield locations of the first, second, third, and fourth elements of each packed pixel type are illustrated in the diagrams below. Each bitfield is interpreted as an unsigned integer value. If the base GL type is supported with more than the minimum precision (e.g. a 9-bit byte) the packed elements are right-justified in the pixel.

```
UNSIGNED_BYTE_3_3_2_EXT:

      7   6   5   4   3   2   1   0
    +-----------+-----------+-------+
    |           |           |       |
    +-----------+-----------+-------+

       first       second     third
       element     element    element


UNSIGNED_SHORT_4_4_4_4_EXT:

      15  14  13  12  11  10  9   8   7   6   5   4   3   2   1   0
    +---------------+---------------+---------------+---------------+
    |               |               |               |               |
    +---------------+---------------+---------------+---------------+

        first           second          third           fourth
        element         element         element         element


UNSIGNED_SHORT_5_5_5_1_EXT:

      15  14  13  12  11  10  9   8   7   6   5   4   3   2   1   0
    +-------------------+-------------------+-------------------+---+
    |                   |                   |                   |   |
    +-------------------+-------------------+-------------------+---+

        first                second              third           fourth
        element              element             element         element


UNSIGNED_INT_8_8_8_8_EXT:

   31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
  +-----------------------+-----------------------+-----------------------+-----------------------+
  |                       |                       |                       |                       |
  +-----------------------+-----------------------+-----------------------+-----------------------+

        first                   second                  third                   fourth
        element                 element                 element                 element


UNSIGNED_INT_10_10_10_2_EXT:

   31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
  +-----------------------------+-----------------------------+-----------------------------+-----+
  |                             |                             |                             |     |
  +-----------------------------+-----------------------------+-----------------------------+-----+

         first                        second                        third               fourth
         element                      element                       element             element
```

The assignment of elements to fields in the packed pixel is as described in the table below:

Format	First Element	Second Element	Third Element	Fourth Element
RGB	red	green	blue	
RGBA	red	green	blue	alpha
ABGR_EXT	alpha	blue	green	red
CMYK_EXT	cyan	magenta	yellow	black

Byte swapping, if enabled, is performed before the elements are extracted from each pixel. The above discussions of row length and image extraction are valid for packed pixels, if "group" is substituted for "element" and the number of elements per group is understood to be one.

Calling DrawPixels with a <type> of BITMAP is a special case in which the data are a series of GL ubyte values. Each ubyte value specifies 8 1-bit elements with its 8 least-significant bits. The 8 single-bit elements are ordered from most significant to least significant if the value of UNPACK_LSB_FIRST is FALSE; otherwise, the ordering is from least significant to most significant. The values of bits other than the 8 least significant in each ubyte are not significant.

The first element of the first row is the first bit (as defined above) of the ubyte pointed to by the pointer passed to DrawPixels. The first element of the second row is the first bit (again as defined above) of the ubyte at location p+k, where k is computed as

```
    k = a * ceiling(nl/8a)
```

There is a mechanism for selecting a sub-rectangle of elements from a BITMAP image as well. Before obtaining the first element from memory, the pointer supplied to DrawPixels is effectively advanced by

```
    UNPACK_SKIP_ROWS * k
```

ubytes. Then UNPACK_SKIP_PIXELS 1-bit elements are ignored, and the subsequent <width> 1-bit elements are obtained, without advancing the ubyte pointer, after which the pointer is advanced by k ubytes. <height> sets of <width> elements are obtained this way.

Conversion to floating-point

This step applies only to groups of components. It is not performed on indices. Each element in a group is converted to a floating-point value according to the appropriate formula in Table 2.4 (section 2.12). Unsigned integer bitfields extracted from packed pixels are interpreted using the formula

```
    f = c / ((2**N)-1)
```

where c is the value of the bitfield (interpreted as an unsigned integer), N is the number of bits in the bitfield, and the division is performed in floating point.

[End of changes to Section 3.6.3]

If this extension is supported, all commands that accept pixel data also accept packed pixel data. These commands are DrawPixels, TexImage1D, TexImage2D, TexImage3DEXT, TexSubImage1DEXT, TexSubImage2DEXT, TexSubImage3DEXT, ConvolutionFilter1DEXT, ConvolutionFilter2DEXT, ConvolutionFilter3DEXT, SeparableFilter2DEXT, SeparableFilter3DEXT, ColorTableSGI, TexImage4DSGIS, and TexSubImage4DSGIS.

Additions to Chapter 4 of the 1.0 Specification (Per-Fragment Operations and the Framebuffer)

[Make the following changes to Section 4.3.2 (Reading Pixels):]

Final Conversion

For an index, if the <type> is not FLOAT, final conversion consists of masking the index with the value given in Table 4.6; if the <type> is FLOAT, then the integer index is converted to a GL float data value. For a component, each component is first clamped to [0,1]. Then, the appropriate conversion formula from Table 4.7 is applied to the component.

<type> Parameter	Index Mask
UNSIGNED_BYTE	2**8 - 1
BITMAP	1
BYTE	2**7 - 1
UNSIGNED_SHORT	2**16 - 1
SHORT	2**15 - 1
UNSIGNED_INT	2**32 - 1
INT	2**31 - 1

Table 4.6: Index masks used by ReadPixels. Floating point data are not masked.

<type> Parameter	GL Data Type	Component Conversion Formula
UNSIGNED_BYTE	ubyte	c = ((2**8)-1)*f
BYTE	byte	c = (((2**8)-1)*f-1)/2
UNSIGNED_SHORT	ushort	c = ((2**16)-1)*f
SHORT	short	c = (((2**16)-1)*f-1)/2
UNSIGNED_INT	uint	c = ((2**32)-1)*f
INT	int	c = (((2**32)-1)*f-1)/2
FLOAT	float	c = f
UNSIGNED_BYTE_3_3_2_EXT	ubyte	c = ((2**N)-1)*f
UNSIGNED_SHORT_4_4_4_4_EXT	ushort	c = ((2**N)-1)*f
UNSIGNED_SHORT_5_5_5_1_EXT	ushort	c = ((2**N)-1)*f
UNSIGNED_INT_8_8_8_8_EXT	uint	c = ((2**N)-1)*f
UNSIGNED_INT_10_10_10_2_EXT	uint	c = ((2**N)-1)*f

Table 4.7: Reversed component conversions - used when component data are being returned to client memory. Color, normal, and depth components are converted from the internal floating-point representation (f) to a datum of the specified GL data type (c) using the equations in this table. All arithmetic is done in the internal floating point format. These conversions apply to component data returned by GL query commands and to components of pixel data returned to client memory. The equations remain the same even if the implemented ranges of the GL data types are greater than the minimum required ranges. (Refer to table 2.2.) Equations with N as the exponent are performed for each bitfield of the packed data type, with N set to the number of bits in the bitfield.

Placement in Client Memory

Groups of elements are placed in memory just as they are taken from memory for DrawPixels. That is, the ith group of the jth row (corresponding to the ith pixel in the jth row) is placed in memory must where the ith group of the jth row would be taken from for DrawPixels. See Unpacking under section 3.6.3. The only difference is that the storage mode parameters whose names begin with PACK_ are used instead of those whose names begin with UNPACK_.

[End of changes to Section 4.3.2]

If this extension is supported, all commands that return pixel data also return packed pixel data. These commands are ReadPixels, GetTexImage, GetHistogramEXT, GetMinmaxEXT, GetConvolutionFilterEXT, GetSeparableFilterEXT, and GetColorTableSGI.

Additions to Chapter 5 of the 1.0 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.0 Specification (State and State Requests)

None

Additions to the GLX Specification

None

GLX Protocol

None

Dependencies on EXT_abgr

If EXT_abgr is not implemented, then the references to ABGR_EXT in this file are invalid, and should be ignored.

Dependencies on EXT_texture3D

If EXT_texture3D is not implemented, then the references to TexImage3DEXT in this file are invalid, and should be ignored.

Dependencies on EXT_subtexture

If EXT_subtexture is not implemented, then the references to TexSubImage1DEXT, TexSubImage2DEXT, and TexSubImage3DEXT in this file are invalid, and should be ignored.

Dependencies on EXT_histogram

If EXT_histogram is not implemented, then the references to GetHistogramEXT and GetMinmaxEXT in this file are invalid, and should be ignored.

Dependencies on EXT_convolution

If EXT_convolution is not implemented, then the references to ConvolutionFilter1DEXT, ConvolutionFilter2DEXT, ConvolutionFilter3DEXT, GetConvolutionFilterEXT, SeparableFilter2DEXT, SeparableFilter3DEXT, and GetSeparableFilterEXT in this file are invalid, and should be ignored.

Dependencies on SGI_color_table

If SGI_color_table is not implemented, then the references to ColorTableSGI and GetColorTableSGI in this file are invalid, and should be ignored.

Dependencies on SGIS_texture4D

If SGIS_texture4D is not implemented, then the references to TexImage4DSGIS and TexSubImage4DSGIS in this file are invalid, and should be ignored.

Dependencies on EXT_cmyka

If EXT_cmyka is not implemented, then the references to CMYK_EXT and CMYKA_EXT in this file are invalid, and should be ignored.

Errors

[For the purpose of this enumeration of errors, GenericPixelFunction represents any OpenGL function that accepts or returns pixel data, using parameters <type> and <format> to define the type and format of that data. Currently these functions are DrawPixels, ReadPixels, TexImage1D, TexImage2D, GetTexImage, TexImage3DEXT, TexSubImage1DEXT, TexSubImage2DEXT, TexSubImage3DEXT, GetHistogramEXT, GetMinmaxEXT, ConvolutionFilter1DEXT, ConvolutionFilter2DEXT, ConvolutionFilter3DEXT, GetConvolutionFilterEXT, SeparableFilter2DEXT, SeparableFilter3DEXT, GetSeparableFilterEXT, ColorTableSGI, GetColorTableSGI, TexImage4DSGIS, and TexSubImage4DSGIS.]

INVALID_OPERATION is generated by GenericPixelFunction if its <type> parameter is UNSIGNED_BYTE_3_3_2_EXT and its <format> parameter does not specify three components. Currently the only 3-component format is RGB.

INVALID_OPERATION is generated by GenericPixelFunction if its <type> parameter is UNSIGNED_SHORT_4_4_4_4_EXT, UNSIGNED_SHORT_5_5_5_1_EXT, UNSIGNED_INT_8_8_8_8_EXT, or UNSIGNED_INT_10_10_10_2_EXT and its <format> parameter does not specify four components. Currently the only 4-component formats are RGBA, ABGR_EXT, and CMYK_EXT.

New State

None

New Implementation Dependent State

None

Name

    EXT_paletted_texture

Name Strings

    GL_EXT_paletted_texture

Version

    $Date: 2004/03/24 01:07:42 $ $Revision: 1.4 $

Number

    78

Support

```
    Intel 810/815.

    Mesa.

    Microsoft software OpenGL implementation.

    Selected NVIDIA GPUs: NV1x (GeForce 256, GeForce2, GeForce4 MX,
    GeForce4 Go, Quadro, Quadro2), NV2x (GeForce3, GeForce4 Ti,
    Quadro DCC, Quadro4 XGL), and NV3x (GeForce FX 5xxxx, Quadro FX
    1000/2000/3000).  NV3 (Riva 128) and NV4 (TNT, TNT2) GPUs and NV4x
    GPUs do NOT support this functionality (no hardware support).
    Future NVIDIA GPU designs will no longer support paletted textures.

    S3 ProSavage, Savage 2000.

    3Dfx Voodoo3, Voodoo5.

    3Dlabs GLINT.
```

Dependencies

```
    GL_EXT_paletted_texture shares routines and enumerants with
    GL_SGI_color_table with the minor modification that EXT replaces SGI.
    In all other ways these calls should function in the same manner and the
    enumerant values should be identical.  The portions of
    GL_SGI_color_table that are used are:

                ColorTableSGI, GetColorTableSGI, GetColorTableParameterivSGI,
                GetColorTableParameterfvSGI.
                COLOR_TABLE_FORMAT_SGI, COLOR_TABLE_WIDTH_SGI,
                COLOR_TABLE_RED_SIZE_SGI, COLOR_TABLE_GREEN_SIZE_SGI,
                COLOR_TABLE_BLUE_SIZE_SGI, COLOR_TABLE_ALPHA_SIZE_SGI,
                COLOR_TABLE_LUMINANCE_SIZE_SGI, COLOR_TABLE_INTENSITY_SIZE_SGI.
    Portions of GL_SGI_color_table which are not used in
    GL_EXT_paletted_texture are:

                CopyColorTableSGI, ColorTableParameterivSGI,
                ColorTableParameterfvSGI.
                COLOR_TABLE_SGI, POST_CONVOLUTION_COLOR_TABLE_SGI,
                POST_COLOR_MATRIX_COLOR_TABLE_SGI, PROXY_COLOR_TABLE_SGI,
                PROXY_POST_CONVOLUTION_COLOR_TABLE_SGI,
                PROXY_POST_COLOR_MATRIX_COLOR_TABLE_SGI, COLOR_TABLE_SCALE_SGI,
                COLOR_TABLE_BIAS_SGI.

    EXT_paletted_texture can be used in conjunction with EXT_texture3D.
    EXT_paletted_texture modifies TexImage3DEXT to accept paletted image
    data and allows TEXTURE_3D_EXT and PROXY_TEXTURE_3D_EXT to be used a
    targets in the color table routines.  If EXT_texture3D is unsupported
    then references to 3D texture support in this spec are invalid and
    should be ignored.

    EXT_paletted_texture can be used in conjunction with
    ARB_texture_cube_map.  EXT_paletted_texture modifies TexImage2D
    to accept paletted image data and allows TEXTURE_CUBE_MAP_ARB, and
    PROXY_TEXTURE_CUBE_MAP_ARB to be used a targets in the color table
    routines.  If ARB_texture_cube_map is unsupported then references
    to cube map texture support in this spec are invalid and should be
    ignored.
```

Overview

```
    EXT_paletted_texture defines new texture formats and new calls to
    support the use of paletted textures in OpenGL.  A paletted texture is
    defined by giving both a palette of colors and a set of image data which
    is composed of indices into the palette.  The paletted texture cannot
    function properly without both pieces of information so it increases the
    work required to define a texture.  This is offset by the fact that the
    overall amount of texture data can be reduced dramatically by factoring
    redundant information out of the logical view of the texture and placing
    it in the palette.

    Paletted textures provide several advantages over full-color textures:

    * As mentioned above, the amount of data required to define a
    texture can be greatly reduced over what would be needed for full-color
    specification.  For example, consider a source texture that has only 256
    distinct colors in a 256 by 256 pixel grid.  Full-color representation
    requires three bytes per pixel, taking 192K of texture data.  By putting
    the distinct colors in a palette only eight bits are required per pixel,
    reducing the 192K to 64K plus 768 bytes for the palette.  Now add an
    alpha channel to the texture.  The full-color representation increases
    by 64K while the paletted version would only increase by 256 bytes.
    This reduction in space required is particularly important for hardware
    accelerators where texture space is limited.

    * Paletted textures allow easy reuse of texture data for images
    which require many similar but slightly different colored objects.
    Consider a driving simulation with heavy traffic on the road.  Many of
    the cars will be similar but with different color schemes.  If
    full-color textures are used a separate texture would be needed for each
```

color scheme, while paletted textures allow the same basic index data to be reused for each car, with a different palette to change the final colors.

* Paletted textures also allow use of all the palette tricks developed for paletted displays. Simple animation can be done, along with strobing, glowing and other palette-cycling effects. All of these techniques can enhance the visual richness of a scene with very little data.

New Procedures and Functions

```
    void ColorTableEXT(
        enum target,
        enum internalFormat,
        sizei width,
        enum format,
        enum type,
        const void *data);

    void ColorSubTableEXT(
        enum target,
        sizei start,
        sizei count,
        enum format,
        enum type,
        const void *data);

    void GetColorTableEXT(
        enum target,
        enum format,
        enum type,
        void *data);

    void GetColorTableParameterivEXT(
        enum target,
        enum pname,
        int *params);

    void GetColorTableParameterfvEXT(
        enum target,
        enum pname,
        float *params);
```

New Tokens

```
    Accepted by the internalformat parameter of TexImage1D, TexImage2D and
    TexImage3DEXT:
        COLOR_INDEX1_EXT                0x80E2
        COLOR_INDEX2_EXT                0x80E3
        COLOR_INDEX4_EXT                0x80E4
        COLOR_INDEX8_EXT                0x80E5
        COLOR_INDEX12_EXT               0x80E6
        COLOR_INDEX16_EXT               0x80E7
    Accepted by the pname parameter of GetColorTableParameterivEXT and
    GetColorTableParameterfvEXT:
        COLOR_TABLE_FORMAT_EXT          0x80D8
        COLOR_TABLE_WIDTH_EXT           0x80D9
        COLOR_TABLE_RED_SIZE_EXT        0x80DA
        COLOR_TABLE_GREEN_SIZE_EXT      0x80DB
        COLOR_TABLE_BLUE_SIZE_EXT       0x80DC
        COLOR_TABLE_ALPHA_SIZE_EXT      0x80DD
        COLOR_TABLE_LUMINANCE_SIZE_EXT  0x80DE
        COLOR_TABLE_INTENSITY_SIZE_EXT  0x80DF

    Accepted by the value parameter of GetTexLevelParameter{if}v:
        TEXTURE_INDEX_SIZE_EXT          0x80ED

    Accepted by the target parameter of ColorTableEXT,
    GetColorTableParameterivEXT, and GetColorTableParameterfvEXT:
        TEXTURE_1D                      0x0DE0
        TEXTURE_2D                      0x0DE1
        TEXTURE_3D_EXT                  0x806F
        TEXTURE_CUBE_MAP_ARB            0x8513
        PROXY_TEXTURE_1D                0x8063
        PROXY_TEXTURE_2D                0x8064
        PROXY_TEXTURE_3D_EXT            0x8070
        PROXY_TEXTURE_CUBE_MAP_ARB      0x851B

    Accepted by the target parameter of ColorSubTableEXT and
    GetColorTableEXT:
        TEXTURE_1D                      0x0DE0
        TEXTURE_2D                      0x0DE1
        TEXTURE_3D_EXT                  0x806F
        TEXTURE_CUBE_MAP_ARB            0x8513
```

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

None

Additions to Chapter 3 of the GL Specification (Rasterization)

Section 3.6.4, 'Pixel Transfer Operations,' subsection 'Color Index Lookup,'

Point two is modified from 'The groups will be loaded as an image into texture memory' to 'The groups will be loaded as an image into texture memory and the internalformat parameter is not one of the color index formats from table 3.8.'

Section 3.8, 'Texturing,' subsection 'Texture Image Specification' is modified as follows:

The portion of the first paragraph discussing interpretation of format, type and data is split from the portion discussing target, width and height. The target, width and height section now ends with the sentence 'Arguments width and height specify the image's width and height.'

The format, type and data section is moved under a subheader 'Direct Color Texture Formats' and begins with 'If internalformat is not one of the color index formats from table 3.8,' and continues with the existing text through the internalformat discussion.

After that section, a new section 'Paletted Texture Formats' has the text:

If format is given as COLOR_INDEX then the image data is composed of integer values representing indices into a table of colors rather than colors themselves. If internalformat is given as one of the color index formats from table 3.8 then the texture will be stored internally as indices rather than undergoing index-to-RGBA mapping as would previously have occurred. In this case the only valid values for type are BYTE, UNSIGNED_BYTE, SHORT, UNSIGNED_SHORT, INT and UNSIGNED_INT.

The image data is unpacked from memory exactly as for a DrawPixels command with format of COLOR_INDEX for a context in color index mode. The data is then stored in an internal format derived from internalformat. In this case the only legal values of internalformat are COLOR_INDEX1_EXT, COLOR_INDEX2_EXT, COLOR_INDEX4_EXT, COLOR_INDEX8_EXT, COLOR_INDEX12_EXT and COLOR_INDEX16_EXT and the internal component resolution is picked according to the index resolution specified by internalformat. Any excess precision in the data is silently truncated to fit in the internal component precision.

An application can determine whether a particular implementation supports a particular paletted format (or any paletted formats at all) by attempting to use the paletted format with a proxy target. TEXTURE_INDEX_SIZE_EXT will be zero if the implementation cannot support the texture as given.

An application can determine an implementation's desired format for a particular paletted texture by making a TexImage call with COLOR_INDEX as the internalformat, in which case target must be a proxy target. After the call the application can query TEXTURE_INTERNAL_FORMAT to determine what internal format the implementation suggests for the texture image parameters. TEXTURE_INDEX_SIZE_EXT can be queried after such a call to determine the suggested index resolution numerically. The index resolution suggested by the implementation does not have to be as large as the input data precision. The resolution may also be zero if the implementation is unable to support any paletted format for the given texture image.

Table 3.8 should be augmented with a column titled 'Index bits.' All existing formats have zero index bits. The following formats are added with zeroes in all existing columns:

Name	Index bits
COLOR_INDEX1_EXT	1
COLOR_INDEX2_EXT	2
COLOR_INDEX4_EXT	4
COLOR_INDEX8_EXT	8
COLOR_INDEX12_EXT	12
COLOR_INDEX16_EXT	16

```
    At the end of the discussion of level the following text should be
    added:

      All mipmapping levels share the same palette.  If levels
      are created with different precision indices then their internal formats
      will not match and the texture will be inconsistent, as discussed above.

    In the discussion of internalformat for CopyTexImage{12}D, at end of the
    sentence specifying that 1, 2, 3 and 4 are illegal there should also be
    a mention that paletted internalformat values are illegal.

    At the end of the width, height, format, type and data section under
    TexSubImage there should be an additional sentence:

      If the target texture has an color index internal format
      then format may only be COLOR_INDEX.

    At the end of the first paragraph describing TexSubImage and
    CopyTexSubImage the following sentence should be added:

      If the target of a CopyTexSubImage is a paletted texture
      image then INVALID_OPERATION is returned.

    After the Alternate Image Specification Commands section, a new 'Palette
    Specification Commands' section should be added.

      Paletted textures require palette information to
      translate indices into full colors.  The command

        void ColorTableEXT(enum target, enum internalformat, sizei width,
                enum format, enum type, const void *data);

      is used to specify the format and size of the palette for paletted
      textures.  target specifies which texture is to have its palette
      changed and may be one of TEXTURE_1D, TEXTURE_2D, PROXY_TEXTURE_1D,
      PROXY_TEXTURE_2D, TEXTURE_3D_EXT, PROXY_TEXTURE_3D_EXT,
      TEXTURE_CUBE_MAP_ARB, or PROXY_TEXTURE_CUBE_MAP_ARB.  internalformat
      specifies the desired format and resolution of the palette when
      in its internal form.  internalformat can be any of the non-index
      values legal for TexImage internalformat although implementations
      are not required to support palettes of all possible formats.
      width controls the size of the palette and must be a power of two
      greater than or equal to one.  format and type specify the number
      of components and type of the data given by data.  format can be
      any of the formats legal for DrawPixels although implementations
      are not required to support all possible formats.  type can be
      any of the types legal for DrawPixels except GL_BITMAP.
      Data is taken from memory and converted just as if each
      palette entry were a single pixel of a 1D texture.  Pixel unpacking and
      transfer modes apply just as with texture data.  After unpacking and
      conversion the data is translated into a internal format that matches
      the given format as closely as possible.  An implementation does not,
      however, have a responsibility to support more than one precision for
      the base formats.

      If the palette's width is greater than than the range of
      the color indices in the texture data then some of the palettes entries
```

will be unused. If the palette's width is less than the range of the color indices in the texture data then the most-significant bits of the texture data are ignored and only the appropriate number of bits of the index are used when accessing the palette.

Specifying a proxy target causes the proxy texture's palette to be resized and its parameters set but no data is transferred or accessed. If an implementation cannot handle the palette data given in the call then the color table width and component resolutions are set to zero.

Portions of the current palette can be replaced with

```
void ColorSubTableEXT(enum target, sizei start, sizei count,
        enum format, enum type, const void *data);
```

target can be any of the non-proxy values legal for ColorTableEXT. start and count control which entries of the palette are changed out of the range allowed by the internal format used for the palette indices. count is silently clamped so that all modified entries all within the legal range. format and type can be any of the values legal for ColorTableEXT. The data is treated as a 1D texture just as in ColorTableEXT.

In the 'Texture State and Proxy State' section the sentence fragment beginning 'six integer values describing the resolutions...' should be changed to refer to seven integer values, with the seventh being the index resolution.

Palette data should be added in as a third category of texture state.

After the discussion of properties, the following should be added:

Next there is the texture palette. All textures have a palette, even if their internal format is not color index. A texture's palette is initially one RGBA element with all four components set to 1.0.

The sentence mentioning that proxies do not have image data or properties should be extended with 'or palettes.'

The sentence beginning 'If the texture array is too large' describing the effects of proxy failure should change to read:

If the implementation is unable to handle the texture image data the proxy width, height, border width and component resolutions are set to zero. This situation can occur when the texture array is too large or an unsupported paletted format was requested.

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the GL Specification (Special Functions)

Section 5.4, 'Display Lists' is modified as follows:

Include PROXY_TEXTURE_1D, PROXY_TEXTURE_2D, PROXY_TEXTURE_3D, and PROXY_TEXTURE_CUBE_MAP_ARB in the list of tokens for which ColorTableEXT is executed immediately.

Additions to Chapter 6 of the GL Specification (State and State Requests)

In the section on GetTexImage, the sentence saying 'The components are assigned among R, G, B and A according to' should be changed to be

If the internal format of the texture is not a color index format then the components are assigned among R, G, B, and A according to Table 6.1. Specifying COLOR_INDEX for format in this case will generate the error INVALID_ENUM. If the internal format of the texture is color index then the components are handled in one of two ways depending on the value of format. If format is not COLOR_INDEX, the texture's indices are passed through the texture's palette and the resulting components are assigned among R, G, B, and A according to Table 6.1. If format is COLOR_INDEX then the data is treated as single components and the palette indices are returned. Components are taken starting...

Following the GetTexImage section there should be a new section:

GetColorTableEXT is used to get the current texture palette.

```
void GetColorTableEXT(enum target, enum format, enum type, void *data);
```

GetColorTableEXT retrieves the texture palette of the texture given by target. target can be any of the non-proxy targets valid for ColorTableEXT. format and type are interpreted just as for ColorTableEXT. All textures have a palette by default so GetColorTableEXT will always be able to return data even if the internal format of the texture is not a color index format.

Palette parameters can be retrieved using

```
void GetColorTableParameterivEXT(enum target, enum pname, int *params);
void GetColorTableParameterfvEXT(enum target, enum pname, float *params);
```

target specifies the texture being queried and pname controls which parameter value is returned. Data is returned in the memory pointed to by params.

Querying COLOR_TABLE_FORMAT_EXT returns the internal format requested by the most recent ColorTableEXT call or the default. COLOR_TABLE_WIDTH_EXT returns the width of the current palette. COLOR_TABLE_RED_SIZE_EXT, COLOR_TABLE_GREEN_SIZE_EXT, COLOR_TABLE_BLUE_SIZE_EXT and COLOR_TABLE_ALPHA_SIZE_EXT return the actual size of the components used to store the palette data internally, not the size requested when the palette was defined.

    Table 6.11, "Texture Objects" should have a line appended for
    TEXTURE_INDEX_SIZE_EXT:

```
TEXTURE_INDEX_SIZE_EXT  n x Z+  GetTexLevelParameter 0 xD texture image i's index resolution 3.8 -
```

New State

    In table 6.16, Texture Objects, p. 224, add the following:

Get Value	Type	Get Command	Initial Value	Description	Sec.	Attribute
TEXTURE_1D	I	GetColorTableEXT	empty	1D palette	3.8	-
TEXTURE_2D	I	GetColorTableEXT	empty	2D palette	3.8	-
TEXTURE_3D	I	GetColorTableEXT	empty	3D palette	3.8	-
TEXTURE_CUBE_MAP	I	GetColorTableEXT	empty	cube map palette	3.8	-
COLOR_TABLE_FORMAT_EXT	2x4xZn	GetColorTableParameterivEXT	RGBA	paletted texture formats	3.8	-
COLOR_TABLE_WIDTH_EXT	2x4xZ+	GetColorTableParameteriv	0	paletted texture width	3.8	-
COLOR_TABLE_x_SIZE_EXT	6x2x4xZ+	GetColorTableParameteriv	0	paletted texture component sizes	3.8	-
TEXTURE_INDEX_SIZE_EXT	nxZ+	GetTexLevelParameter	0	texture image's index resolution	3.8	-

New Implementation Dependent State

    None

Revision History

Original draft, revision 0.5, December 20, 1995 (drewb) Created

Minor revisions and clarifications, revision 0.6, January 2, 1996 (drewb)
    Replaced all request-for-comment blocks with final text
    based on implementation.

Minor revisions and clarifications, revision 0.7, Feburary 5, 1996 (drewb)
    Specified the state of the palette color information
    when existing data is replaced by new data.

    Clarified behavior of TexPalette on inconsistent textures.

Major changes due to ARB review, revision 0.8, March 1, 1996 (drewb)
    Switched from using TexPaletteEXT and GetTexPaletteEXT
    to using SGI's ColorTableEXT routines. Added ColorSubTableEXT so
    equivalent functionality is available.

    Allowed proxies in all targets.

    Changed PALETTE?_EXT values to COLOR_INDEX?_EXT. Added
    support for one and two bit palettes. Removed PALETTE_INDEX_EXT in
    favor of COLOR_INDEX.

    Decoupled palette size from texture data type. Palette
    size is controlled only by ColorTableEXT.

Changes due to ARB review, revision 1.0, May 23, 1997 (drewb)
    Mentioned texture3D.

    Defined TEXTURE_INDEX_SIZE_EXT.

```
        Allowed implementations to return an index size of zero to indicate
        no support for a particular format.

        Allowed usage of GL_COLOR_INDEX as a generic format in
        proxy queries for determining an optimal index size for a particular
        texture.

        Disallowed CopyTexImage and CopyTexSubImage to paletted
        formats.

        Deleted mention of index transfer operations during GetTexImage with
        paletted formats.

    Changes due to ARB_texture_cube_map, revision 1.1, June 27, 2002.
        Add language to section 5.4 about proxy texture tokens for ColorTable
        executing immediately.

        Document ARB_texture_cube_map interactions.

        Document texture target usage for ColorTable API.

        Add "New State" section with table and "New Implementation Dependent
        State" sections.

    Changes, revision 1.4, March 24, 2004.
        Document vendor support for this extension; note that future NVIDIA
        GPU designs will not support this extension.
```

Name

    EXT_pixel_buffer_object

Name Strings

    GL_EXT_pixel_buffer_object

Status

    Implemented by NVIDIA drivers (Release 55).

IP Status

    Unknown.

Version

    NVIDIA Date: March 29, 2004 (version 1.0)

Number

    302

Status

    NVIDIA Release 55 (early 2004) drivers support this extension.

Dependencies

    Written based on the wording of the OpenGL 1.5 specification.

    GL_NV_pixel_data_range affects the definition of this extension.

Overview

    This extension expands on the interface provided by buffer objects.
    It is intended to permit buffer objects to be used not only with
    vertex array data, but also with pixel data.
    Buffer objects were promoted from the ARB_vertex_buffer_object
    extension in OpenGL 1.5.

    Recall that buffer objects conceptually are nothing more than arrays
    of bytes, just like any chunk of memory. Buffer objects allow GL
    commands to source data from a buffer object by binding the buffer
    object to a given target and then overloading a certain set of GL
    commands' pointer arguments to refer to offsets inside the buffer,
    rather than pointers to user memory. An offset is encoded in a
    pointer by adding the offset to a null pointer.

    This extension does not add any new functionality to buffer
    objects themselves. It simply adds two new targets to which buffer
    objects can be bound: PIXEL_PACK_BUFFER and PIXEL_UNPACK_BUFFER.
    When a buffer object is bound to the PIXEL_PACK_BUFFER target,
    commands such as ReadPixels write their data into a buffer object.
    When a buffer object is bound to the PIXEL_UNPACK_BUFFER target,
    commands such as DrawPixels read their data from a buffer object.

There are a wide variety of applications for such functionality.
Some of the most interesting ones are:

- "Render to vertex array." The application can use a fragment program to render some image into one of its buffers, then read this image out into a buffer object via ReadPixels. Then, it can use this buffer object as a source of vertex data.

- Streaming textures. If the application uses MapBuffer/UnmapBuffer to write its data for TexSubImage into a buffer object, at least one of the data copies usually required to download a texture can be eliminated, significantly increasing texture download performance.

- Asynchronous ReadPixels. If an application needs to read back a number of images and process them with the CPU, the existing GL interface makes it nearly impossible to pipeline this operation. The driver will typically send the hardware a readback command when ReadPixels is called, and then wait for all of the data to be available before returning control to the application. Then, the application can either process the data immediately or call ReadPixels again; in neither case will the readback overlap with the processing. If the application issues several readbacks into several buffer objects, however, and then maps each one to process its data, then the readbacks can proceed in parallel with the data processing.

Issues

How does this extension relate to ARB_vertex_buffer_object?

It builds on the ARB_vertex_buffer_object framework by adding two new targets that buffers can be bound to.

How does this extension relate to NV_pixel_data_range?

This extension relates to NV_pixel_data_range in the same way that ARB_vertex_buffer_object relates to NV_vertex_array_range. To paraphrase the ARB_vertex_buffer_object spec, here are the main differences:

- Applications are no longer responsible for memory management and synchronization.

- Applications may still access high-performance memory directly, but this is optional, and such access is more restricted.

- Buffer changes (BindBuffer) are generally expected to be very lightweight, rather than extremely heavyweight (PixelDataRangeNV).

- A platform-specific allocator such as wgl/glXAllocateMemoryNV is no longer required.

Can a given buffer be used for both vertex and pixel data?

RESOLVED: YES. All buffers can be used with all buffer bindings, in whatever combinations the application finds useful. Consider yourself warned, however, by the following issue.

May implementations make use of the target as a hint to select an appropriate memory space for the buffer?

RESOLVED: YES, as long as such behavior is transparent to the application. Some implementations may choose, for example, that they would rather stream vertex data from write-combined system memory, element (or index) data from video memory, and pixel data from video memory.

In fact, one can imagine arbitrarily complicated heuristics for selecting the memory space, based on factors such as the target, the "usage" argument, and the application's observed behavior.

While it is entirely legal to create a buffer object by binding it to ARRAY_BUFFER and loading it with data, then using it with the PIXEL_UNPACK_BUFFER_EXT or PIXEL_PACK_BUFFER_EXT binding, such behavior is liable to confuse the driver and may hurt performance. If the driver implemented the hypothetical heuristic described earlier, such a buffer might have already been located in write-combined system memory, and so the driver would have to choose between two bad options: relocate the buffer into video memory, or accept lower performance caused by streaming pixel data from slower system memory.

Should all pixel path commands be supported, or just a subset of them?

RESOLVED: ALL. While there is little reason to believe that, say, ConvolutionFilter2D would benefit from this extension, there is no reason _not_ to support it. The full list of commands affected by this extension is listed in the spec.

Should PixelMap and GetPixelMap be supported?

RESOLVED: YES. They're not really pixel path operations, but, again, there is no good reason to omit operations, and they _are_ operations that pass around big chunks of pixel-related data. If we support PolygonStipple, surely we should support this.

How does the buffer binding state push/pop?

RESOLVED: As part of the pixel store client state. This is analogous to how the vertex buffer object bindings pushed/popped as part of the vertex array client state.

Should NV_pixel_data_range (PDR) be used concurrently with pixel buffer objects ?

RESOLVED: NO. While it would be possible to allocate a memory range for PDR, using a pointer into this memory range with one of the commands affected by EXT_pixel_buffer_object will not work if a pixel buffer object other than zero is bound to the buffer binding point affecting the command. Pixel buffer objects always have higher precedence than PDR.

Do the null pointer rules for glTexImage1D, glTexImage2D and glTexImage3D for allocating textures with undefined content also apply when a non-zero buffer object is bound to PIXEL_UNPACK_BUFFER_BINDING_EXT ?

RESOLVED: NO. The null pointer is interpreted as a non-zero pointer to the data storage whose contents may be still undefined. This data will be used to create the texture array. If the null pointer rule is required, no non-zero buffer object should be bound to PIXEL_UNPACK_BUFFER_BINDING_EXT.

New Procedures and Functions

None.

New Tokens

Accepted by the <target> parameters of BindBuffer, BufferData, BufferSubData, MapBuffer, UnmapBuffer, GetBufferSubData, GetBufferParameteriv, and GetBufferPointerv:

PIXEL_PACK_BUFFER_EXT	0x88EB
PIXEL_UNPACK_BUFFER_EXT	0x88EC

Accepted by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

PIXEL_PACK_BUFFER_BINDING_EXT	0x88ED
PIXEL_UNPACK_BUFFER_BINDING_EXT	0x88EF

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.2.1 Specification (Rasterization)

Additions to subsection 3.8.1 of the 1.2.1 Specification (Texture Image Specification)

The extension EXT_pixel_buffer_object makes an exception to this rule of passing a null pointer to glTexImage1D, glTexImage2D and glTexImage3D. If PIXEL_UNPACK_BUFFER_BINDING_EXT is non-zero and a null pointer is passed to these functions, the texture array is created and the image contents are sourced from the data store of the bound buffer object.

Additions to Chapter 4 of the 1.2.1 Specification (Per-Fragment Operations and the Frame Buffer)

Added a subsection 4.3.5 (Pixel Buffer Object unpack operation) in section 4.3 (Drawing, Reading and copying Pixels)

The extension EXT_pixel_buffer_object affects the operation of several OpenGL commands described in section 3.6 (Pixel Rectangles), section 3.7 (Bitmaps), and section 3.8 (Texturing).

In unextended OpenGL 1.3 with ARB_imaging support, the commands glBitmap, glColorSubTable, glColorTable, glCompressedTexImage1D, glCompressedTexImage2D, glCompressedTexImage3D, glCompressedTexSubImage1D, glCompressedTexSubImage2D, glCompressedTexSubImage3D, glConvolutionFilter1D, glConvolutionFilter2D, glDrawPixels, glPixelMapfv, glPixelMapuiv, glPixelMapusv, glPolygonStipple, glSeparableFilter2D, glTexImage1D, glTexImage2D, glTexImage3D, glTexSubImage1D, glTexSubImage2D and glTexSubImage3D operate as previously defined, except that pixel data is sourced from a buffer object's data store if PIXEL_UNPACK_BUFFER_BINDING_EXT is non-zero. When the data is sourced from a buffer object, the pointer value passed in as an argument to the command is used to compute an offset, in basic machine units, into the data store of the buffer object. This offset is computed by subtracting a null pointer from the pointer value, where both pointers are treated as pointers to basic machine units.

Additions to Chapter 5 of the 1.2.1 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.2.1 Specification (State and State Requests)

Additions to subsection 6.1.13 (Buffer Object Queries) in chapter 6

In unextended OpenGL 1.5 with ARB_imaging support, the commands glGetColorTable, glGetCompressedTexImage, glGetConvolutionFilter, glGetHistogram, glGetMinmax, glGetPixelMapfv, glGetPixelMapuiv, glGetPixelMapusv, glGetPolygonStipple, glGetSeparableFilter, glGetTexImage and glReadPixels operate as previously defined, except that pixel data is stored in a buffer object's data store if PIXEL_PACK_BUFFER_BINDING_EXT is non-zero. When a buffer object is the target of the pixel data, the target pointer value passed in as an argument to the command is used to compute an offset, in basic machine units, into the data store of the buffer object. This offset is computed by subtracting a null pointer from the pointer value, where both pointers are treated as pointers to basic machine units.

Errors

None

New State

(table 6.20, Pixels, p. 235)

```
Get Value                        Type  Get Command  Initial Value  Sec     Attribute
---------                        ----  -----------  -------------  ---     ---------
PIXEL_PACK_BUFFER_BINDING_EXT    Z+    GetIntegerv  0              4.3.5   pixel-store
PIXEL_UNPACK_BUFFER_BINDING_EXT  Z+    GetIntegerv  0              6.1.13  pixel-store
```

New Implementation Dependent State

    (none)

Usage Examples

    Convenient macro definition for specifying buffer offsets:

```
        #define BUFFER_OFFSET(i) ((char *)NULL + (i))
```

    Example 1: Render to vertex array

```
        // create a buffer object for a number of vertices consisting of
        // 4 float values per vertex
        GenBuffers(1, vertexBuffer);
        BindBuffer(PIXEL_PACK_BUFFER_EXT, vertexBuffer);
        BufferData(PIXEL_PACK_BUFFER_EXT, numberVertices*4, NULL, DYNAMIC_DRAW);

        // render vertex data into framebuffer using a fragment program
        BindProgramARB(FRAGMENT_PROGRAM_ARB, fragmentProgram);
        DrawBuffer(GL_BACK);
        renderVertexData();
        BindProgramARB(FRAGMENT_PROGRAM_ARB, 0);

        // read the vertex data back from framebuffer
        ReadBuffer(GL_BACK);
        ReadPixels(0, 0, numberVertices*4, height/2,
            GL_BGRA, GL_FLOAT, BUFFER_OFFSET(0));

        // change the binding point of the buffer object to
        // the vertex array binding point
        BindBuffer(GL_ARRAY_BUFFER, vertexBuffer);

        EnableClientState(VERTEX_ARRAY);
        VertexPointer(4, FLOAT, 0, BUFFER_OFFSET(0));
        DrawArrays(TRIANGLE_STRIP, 0, numberVertices);
```

Example 2: Streaming textures

streaming textures using NV_pixel_data_range

```
    void *pdrMemory, *texData;

    pdrMemory = AllocateMemoryNV(texsize, 0.0, 1.0, 1.0);

    PixelDataRangeNV(GL_WRITE_PIXEL_DATA_RANGE_NV, texsize, pdrMemory);

    EnableClientState(GL_WRITE_PIXEL_DATA_RANGE_NV);

    // setup texture environment
    ...

    texData = getNextImage();

    while (texData) {

        memcpy(pdrMemory, texData, texsize);

        FlushPixelDataRangeNV(GL_WRITE_PIXEL_DATA_RANGE_NV);

        TexSubImage2D(GL_TEXTURE_2D, 0, 0, 0,
            texWidth, texHeight, GL_BGRA, GL_UNSIGNED_BYTE, pdrMemory);

        // draw textured geometry
        Begin(GL_QUADS);
        ...
        End();

        texData = getNextImage();
    }

    DisableClientState(GL_WRITE_PIXEL_DATA_RANGE_NV);

    FreeMemoryNV(pdrMemory);
streaming textures using EXT_pixel_buffer_object:

    void *pboMemory, *texData;

    // create and bind texture image buffer object
    GenBuffers(1, &texBuffer);
    BindBuffer(PIXEL_UNPACK_BUFFER_EXT, texBuffer);
    BufferData(PIXEL_UNPACK_BUFFER_EXT, texSize, NULL, STREAM_DRAW);

    texData = getNextImage();

    while (texData) {

        // map the texture image buffer
        pboMemory = MapBuffer(PIXEL_UNPACK_BUFFER_EXT, WRITE_ONLY);

        // modify (sub-)buffer data
        memcpy(pboMemory, texData, texsize);

        // unmap the texture image buffer
        if (!UnmapBuffer(PIXEL_UNPACK_BUFFER_EXT)) {
            // Handle error case
        }

        // update (sub-)teximage from texture image buffer
        TexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, texWidth, texHeight,
                      GL_BGRA, GL_UNSIGNED_BYTE, BUFFER_OFFSET(0));

        // draw textured geometry
        Begin(GL_QUADS);
        ...
        End();

        texData = getNextImage();
    }

    BindBuffer(PIXEL_UNPACK_BUFFER_EXT, 0);
```

Example 3: Asynchronous ReadPixels

traditional ReadPixels

```
    unsigned int readBuffer[imagewidth*imageheight*4];

    // render to framebuffer
    DrawBuffer(GL_BACK);
    renderScene()

    // read image from framebuffer
    ReadBuffer(GL_BACK);
    ReadPixels();

    // process image when ReadPixels returns after reading the whole buffer
    processImage(readBuffer);
```

asynchronous ReadPixels

```
    GenBuffers(2, imageBuffers);

    BindBuffer(PIXEL_PACK_BUFFER_EXT, imageBuffers[0]);
    BufferData(PIXEL_PACK_BUFFER_EXT, imageSize / 2, NULL, STATIC_READ);

    BindBuffer(PIXEL_PACK_BUFFER_EXT, imageBuffers[1]);
    BufferData(PIXEL_PACK_BUFFER_EXT, imageSize / 2, NULL, STATIC_READ);

    // render to framebuffer
    DrawBuffer(GL_BACK);
    renderScene();

    // Bind two different buffer objects and start the ReadPixels
    // asynchronously. Each call will return directly after starting the
    // DMA transfer.
    BindBuffer(PIXEL_PACK_BUFFER_EXT, imageBuffers[0]);
    ReadPixels(0, 0, width, height/2,
        GL_BGRA, GL_UNSIGNED_BYTE, BUFFER_OFFSET(0));

    BindBuffer(PIXEL_PACK_BUFFER_EXT, imageBuffers[1]);
    ReadPixels(0, height/2, width, height/2, GL_BGRA, GL_UNSIGNED_BYTE,
               BUFFER_OFFSET(0));

    // process partial images
    pboMemory1 = MapBuffer(PIXEL_PACK_BUFFER_EXT, READ_ONLY);
    processImage(pboMemory1);
    pboMemory2 = MapBuffer(PIXEL_PACK_BUFFER_EXT, READ_ONLY);
    processImage(pboMemory2);

    // unmap the image buffers
    BindBuffer(PIXEL_PACK_BUFFER_EXT, imageBuffers[0]);
    if (!UnmapBuffer(PIXEL_PACK_BUFFER_EXT)) {
        // Handle error case
    }
    BindBuffer(PIXEL_PACK_BUFFER_EXT, imageBuffers[1]);
    if (!UnmapBuffer(PIXEL_PACK_BUFFER_EXT)) {
        // Handle error case
    }
```

Name

    EXT_point_parameters

Name Strings

    GL_EXT_point_parameters

Version

    $Date: 1997/08/21 21:26:36 $ $Revision: 1.6 $

Number

    54

Dependencies

SGIS_multisample affects the definition of this extension.

Overview

This extension supports additional geometric characteristics of points. It can be used to render particles or tiny light sources, commonly referred as "Light points".

The raster brightness of a point is a function of the point area, point color, point transparency, and the response of the display's electron gun and phosphor. The point area and the point transparency are derived from the point size, currently provided with the <size> parameter of glPointSize.

The primary motivation is to allow the size of a point to be affected by distance attenuation. When distance attenuation has an effect, the final point size decreases as the distance of the point from the eye increases.

The secondary motivation is a mean to control the mapping from the point size to the raster point area and point transparency. This is done in order to increase the dynamic range of the raster brightness of points. In other words, the alpha component of a point may be decreased (and its transparency increased) as its area shrinks below a defined threshold.

This extension defines a derived point size to be closely related to point brightness. The brightness of a point is given by:

```
                        1
    dist_atten(d) = -------------------
                    a + b * d + c * d^2

    brightness(Pe) = Brightness * dist_atten(|Pe|)
```

where 'Pe' is the point in eye coordinates, and 'Brightness' is some initial value proportional to the square of the size provided with glPointSize. Here we simplify the raster brightness to be a function of the rasterized point area and point transparency.

```
                  brightness(Pe)                  brightness(Pe) >= Threshold_Area
    area(Pe) =
                  Threshold_Area                  Otherwise

    factor(Pe) = brightness(Pe)/Threshold_Area

    alpha(Pe) = Alpha * factor(Pe)
```

where 'Alpha' comes with the point color (possibly modified by lighting).

'Threshold_Area' above is in area units. Thus, it is proportional to the square of the threshold provided by the programmer through this extension.

The new point size derivation method applies to all points, while the threshold applies to multisample points only.

Issues

* Does point alpha modification affect the current color ?

  No.

* Do we need a special function glGetPointParameterfvEXT, or get by with glGetFloat ?

  No.

* If alpha is 0, then we could toss the point before it reaches the fragment stage.

  No. This can be achieved with enabling the alpha test with reference of 0 and function of LEQUAL.

* Do we need a disable for applying the threshold ? The default threshold value is 1.0. It is applied even if the point size is constant.

  If the default threshold is not overriden, the area of multisample points with provided constant size of less than 1.0, is mapped to 1.0, while the alpha component is modulated accordingly, to compensate for the larger area. For multisample points this is not a problem, as there are no relevant applications yet. As mentioned above, the threshold does not apply to alias or antialias points.

  The alternative is to have a disable of threshold application, and state that threshold (if not disabled) applies to non antialias points only (that is, alias and multisample points).

  The behavior without an enable/disable looks fine.

* Future extensions (to the extension)

  1. GL_POINT_FADE_ALPHA_CLAMP_EXT

  When the derived point size is larger than the threshold size defined by the GL_POINT_FADE_THRESHOLD_SIZE_EXT parameter, it might be desired to clamp the computed alpha to a minimum value, in order to keep the point visible. In this case the formula below change:

```
        factor = (derived_size/threshold)^2

                        factor                  clamp <= factor
        clamped_value =
                        clamp                   factor < clamp

                 1.0                            derived_size >= threshold
        alpha *=
                 clamped_value                  Otherwise

        where clamp is defined by the GL_POINT_FADE_ALPHA_CLAMP_EXT new
        parameter.
```

New Procedures and Functions

```
    void glPointParameterfEXT ( GLenum pname, GLfloat param );
    void glPointParameterfvEXT ( GLenum pname, GLfloat *params );
```

New Tokens

    Accepted by the <pname> parameter of glPointParameterfEXT, and the <pname>
    of glGet:

        GL_POINT_SIZE_MIN_EXT
        GL_POINT_SIZE_MAX_EXT
        GL_POINT_FADE_THRESHOLD_SIZE_EXT

    Accepted by the <pname> parameter of glPointParameterfvEXT, and the <pname>
    of glGet:

        GL_POINT_SIZE_MIN_EXT   0x8126
        GL_POINT_SIZE_MAX_EXT   0x8127
        GL_POINT_FADE_THRESHOLD_SIZE_EXT   0x8128
        GL_DISTANCE_ATTENUATION_EXT   0x8129

Additions to Chapter 2 of the 1.0 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.0 Specification (Rasterization)

All parameters of the glPointParameterfEXT and glPointParameterfvEXT functions set various values applied to point rendering. The derived point size is defined to be the <size> provided with glPointSize modulated with a distance attenuation factor.

The parameters GL_POINT_SIZE_MIN_EXT and GL_POINT_SIZE_MAX_EXT simply define an upper and lower bounds respectively on the derived point size.

The above parameters affect non multisample points as well as multisample points, while the GL_POINT_FADE_THRESHOLD_SIZE_EXT parameter, has no effect on non multisample points. If the derived point size is larger than the threshold size defined by the GL_POINT_FADE_THRESHOLD_SIZE_EXT parameter, the derived point size is used as the diameter of the rasterized point, and the alpha component is intact. Otherwise, the threshold size is set to be the diameter of the rasterized point, while the alpha component is

```
    modulated accordingly, to compensate for the larger area.

    The distance attenuation function coefficients, namely a, b, and c in:

                          1
        dist_atten(d) = -------------------
                        a + b * d + c * d^2

    are defined by the <pname> parameter GL_DISTANCE_ATTENUATION_EXT of the
    function glPointParameterfvEXT. By default a = 1, b = 0, and c = 0.

    Let 'size' be the point size provided with glPointSize,  let 'dist' be the
    distance of the point from the eye, and let 'threshold' be the threshold
    size defined by the GL_POINT_FADE_THRESHOLD_SIZE parameter of
    glPointParameterfEXT. The derived point size is given by:

        derived_size = size * sqrt(dist_atten(dist))

    Note that when default values are used, the above formula reduces to:

        derived_size = size

    the diameter of the rasterized point is given by:

                   derived_size                derived_size >= threshold
        diameter =
                   threshold                   Otherwise

    The alpha of a point is calculated to allow the fading of points instead of
    shrinking them past a defined threshold size. The alpha component of the
    rasterized point is given by:

                 1                             derived_size >= threshold
        alpha *=
                 (derived_size/threshold)^2    Otherwise

    The threshold defined by GL_POINT_FADE_THRESHOLD_SIZE_EXT is not clamped
    to the minimum and maximum point sizes.

    Points do not affect the current color.

    This extension doesn't change the feedback or selection behavior of points.
```

Additions to Chapter 4 of the 1.0 Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the 1.0 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.0 Specification (State and State Requests)

None

Additions to the GLX Specification

None

Dependencies on SGIS_multisample

If SGIS_multisample is not implemented, then the references to multisample points are invalid, and should be ignored.

Errors

INVALID_ENUM is generated if PointParameterfEXT parameter <pname> is not GL_POINT_SIZE_MIN_EXT, GL_POINT_SIZE_MAX_EXT, or GL_POINT_FADE_THRESHOLD_SIZE_EXT.

INVALID_ENUM is generated if PointParameterfvEXT parameter <pname> is not GL_POINT_SIZE_MIN_EXT, GL_POINT_SIZE_MAX_EXT, GL_POINT_FADE_THRESHOLD_SIZE_EXT, or GL_DISTANCE_ATTENUATION_EXT

INVALID_VALUE is generated when values are out of range according to:

<pname>	valid range
GL_POINT_SIZE_MIN_EXT	>= 0
GL_POINT_SIZE_MAX_EXT	>= 0
GL_POINT_FADE_THRESHOLD_SIZE_EXT	>= 0

Issues

- should we generate INVALID_VALUE or just clamp?

New State

Get Value	Get Command	Type	Initial Value	Attribute
GL_POINT_SIZE_MIN_EXT	GetFloatv	R	0	point
GL_POINT_SIZE_MAX_EXT	GetFloatv	R	M	point
GL_POINT_FADE_THRESHOLD_SIZE_EXT	GetFloatv	R	1	point
GL_DISTANCE_ATTENUATION_EXT	GetFloatv	3xR	(1,0,0)	point

M is the largest available point size.

New Implementation Dependent State

None

Backwards Compatibility

This extension replaces SGIS_point_parameters. The procedures, tokens, and name strings now refer to EXT instead of SGIS. Enumerant values are unchanged. SGI implementations which previously provided this functionality should support both forms of the extension.

Name

EXT_rescale_normal

Name Strings

GL_EXT_rescale_normal

Version

$Date: 1997/07/02 23:38:17 $ $Revision: 1.7 $

Number

27

Dependencies

None

Overview

When normal rescaling is enabled a new operation is added to the transformation of the normal vector into eye coordinates. The normal vector is rescaled after it is multiplied by the inverse modelview matrix and before it is normalized.

The rescale factor is chosen so that in many cases normal vectors with unit length in object coordinates will not need to be normalized as they are transformed into eye coordinates.

New Procedures and Functions

None

New Tokens

Accepted by the <cap> parameter of Enable, Disable, and IsEnabled, and by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

```
    RESCALE_NORMAL_EXT                   0x803A
```

Additions to Chapter 2 of the 1.1 Specification (OpenGL Operation)

Section 2.10.3

Finally, we consider how the ModelView transformation state affects normals. Normals are of interest only in eye coordinates, so the rules governing their transformation to other coordinate systems are not examined.

Normals which have unit length when sent to the GL, have their length changed by the inverse of the scaling factor after transformation by the model-view inverse matrix when the model-view matrix represents a uniform scale. If rescaling is enabled, then normals specified with

```
    the Normal command are rescaled after transformation by the ModelView
    Inverse.

    Normals sent to the GL may or may not have unit length. In addition,
    the length of the normals after transformation might be altered due
    to transformation by the model-view inverse matrix. If normalization
    is enabled, then normals specified with the Normal3 command are
    normalized after transformation by the model-view inverse matrix and
    after rescaling if rescaling is enabled.  Normalization and rescaling
    are controlled with

        void Enable( enum target);

    and

        void Disable( enum target);

    with target equal to NORMALIZE or RESCALE_NORMAL. This requires two
    bits of state.  The initial state is for normals not to be normalized or
    rescaled.
    .
    .
    .

    Therefore, if the modelview matrix is M, then the transformed plane equation
    is

     (n_x' n_y' n_z' q') = ((n_x n_y n_z q) * (M^-1)),

    the rescaled normal is

     (n_x" n_y" n_z")  = f * (n_x' n_y' n_z'),

    and the fully transformed normal is

              1                                  (n_x")
          ____________                           (n_y")             (2.1)
        __________________________________       (n_z")
       V (n_x")^2 + (n_y")^2 + (n_z")^2

     If rescaling is disabled then f is 1, otherwise f is computed
     as follows:

     Let m_ij denote the matrix element in row i and column j of M^-1,
     numbering the topmost row of the matrix as row 1, and the leftmost
     column as column 1. Then

                           1
                       __________
            f =   ________________________________
                 V (m_31)^2 + (m_32)^2 + (m_33)^2

     Alternatively, an implementation my chose to normalize the normal
     instead of rescaling the normal. Then
```

$$f = \frac{1}{\sqrt{(n\_x')^2 + (n\_y')^2 + (n\_z')^2}}$$

If normalization is disabled, then the square root in equation 2.1 is replaced with 1, otherwise . . . .

Additions to Chapter 3 of the 1.1 Specification (Rasterization)

None

Additions to Chapter 4 of the 1.1 Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the 1.1 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.1 Specification (State and State Requests)

None

Additions to the GLX Specification

None

GLX Protocol

None

Errors

None

New State

Get Value	Get Command	Type	Initial Value	Attribute
RESCALE_NORMAL_EXT	IsEnabled	B	FALSE	transform/enable

New Implementation Dependent State

None

```
Name

    EXT_secondary_color

Name Strings

    GL_EXT_secondary_color

Version

    NVIDIA Date: February 22, 2000
    $Date: 1999/06/21 19:57:47 $ $Revision: 1.8 $

Number

    145

Dependencies

    Either EXT_separate_specular_color or OpenGL 1.2 is required, to specify
    the "Color Sum" stage and other handling of the secondary color. This is
    written against the 1.2 specification (available from www.opengl.org).

Overview

    This extension allows specifying the RGB components of the secondary
    color used in the Color Sum stage, instead of using the default
    (0,0,0,0) color. It applies only in RGBA mode and when LIGHTING is
    disabled.

Issues

  * Can we use the secondary alpha as an explicit fog weighting factor?

        ISVs prefer a separate interface (see GL_EXT_fog_coord). The current
        interface specifies only the RGB elements, leaving the option of a
        separate extension for SecondaryColor4() entry points open (thus
        the apparently useless ARRAY_SIZE state entry).

        There is an unpleasant asymmetry with Color3() - one assumes A =
        1.0, the other assumes A = 0.0 - but this appears unavoidable given
        the 1.2 color sum specification language. Alternatively, the color
        sum language could be rewritten to not sum secondary A.

  * What about multiple "color iterators" for use with aggrandized
    multitexture implementations?

        We may need this eventually, but the secondary color is well defined
        and a more generic interface doesn't seem justified now.

  * Interleaved array formats?

        No. The multiplicative explosion of formats is too great.
  * Do we want to be able to query the secondary color value? How does it
    interact with lighting?

        The secondary color is not part of the GL state in the
        separate_specular_color extension that went into OpenGL 1.2. There,
        it can't be queried or obtained via feedback.

        The secondary_color extension is slightly more general-purpose, so
        the secondary color is explicitly in the GL state and can be queried
        - but it's still somewhat limited and can't be obtained via
        feedback, for example.
```

New Procedures and Functions

```
    void SecondaryColor3[bsifd ubusui]EXT(T components)
    void SecondaryColor3[bsifd ubusui]vEXT(T components)
    void SecondaryColorPointerEXT(int size, enum type, sizei stride,
                                  void *pointer)
```

New Tokens

```
    Accepted by the <cap> parameter of Enable, Disable, and IsEnabled,
    and by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv,
    and GetDoublev:

        COLOR_SUM_EXT                       0x8458

    Accepted by the <pname> parameter of GetBooleanv, GetIntegerv,
    GetFloatv, and GetDoublev:

        CURRENT_SECONDARY_COLOR_EXT         0x8459
        SECONDARY_COLOR_ARRAY_SIZE_EXT      0x845A
        SECONDARY_COLOR_ARRAY_TYPE_EXT      0x845B
        SECONDARY_COLOR_ARRAY_STRIDE_EXT    0x845C

    Accepted by the <pname> parameter of GetPointerv:

        SECONDARY_COLOR_ARRAY_POINTER_EXT   0x845D

    Accepted by the <array> parameter of EnableClientState and
    DisableClientState:

        SECONDARY_COLOR_ARRAY_EXT           0x845E
```

Additions to Chapter 2 of the 1.2 Draft Specification (OpenGL Operation)

```
  These changes describe a new current state type, the secondary color, and
  the commands to specify it:

  - (2.6, p. 12) Second paragraph changed to:

        "Each vertex is specified with two, three, or four coordinates. In
        addition, a current normal, current texture coordinates, current
        color, and current secondary color may be used in processing each
        vertex."
```

```
    Third paragraph, second sentence changed to:

        "These associated colors are either based on the current color and
        current secondary color, or produced by lighting, depending on
        whether or not lighting is enabled."

  - 2.6.3, p. 19) First paragraph changed to

        "The only GL commands that are allowed within any Begin/End pairs
        are the commands for specifying vertex coordinates, vertex colors,
        normal coordinates, and texture coordinates (Vertex, Color,
        SecondaryColorEXT, Index, Normal, TexCoord)..."

  - (2.7, p. 20) Starting with the fourth paragraph, change to:

        "Finally, there are several ways to set the current color and
        secondary color. The GL stores a current single-valued color index
        as well as a current four-valued RGBA color and secondary color.
        Either the index or the color and secondary color are significant
        depending as the GL is in color index mode or RGBA mode. The mode
        selection is made when the GL is initialized.

        The commands to set RGBA colors and secondary colors are:

            void Color[34][bsifd ubusui](T components)
            void Color[34][bsifd ubusui]v(T components)
            void SecondaryColor3[bsifd ubusui]EXT(T components)
            void SecondaryColor3[bsifd ubusui]vEXT(T components)

        The color command has two major variants: Color3 and Color4. The
        four value versions set all four values. The three value versions
        set R, G, and B to the provided values; A is set to 1.0. (The
        conversion of integer color components (R, G, B, and A) to
        floating-point values is discussed in section 2.13.)

        The secondary color command has only the three value versions.
        Secondary A is always set to 0.0.

        Versions of the Color and SecondaryColorEXT commands that take
        floating-point values accept values nominally between 0.0 and
        1.0...."

    The last paragraph is changed to read:

        "The state required to support vertex specification consists of four
        floating-point numbers to store the current texture coordinates s,
        t, r, and q, four floating-point values to store the current RGBA
        color, four floating-point values to store the current RGBA
        secondary color, and one floating-point value to store the current
        color index. There is no notion of a current vertex, so no state is
        devoted to vertex coordinates. The initial values of s, t, and r of
        the current texture coordinates are zero; the initial value of q is
        one. The initial current normal has coordinates (0,0,1). The initial
        RGBA color is (R,G,B,A) = (1,1,1,1). The initial RGBA secondary
        color is (R,G,B,A) = (0,0,0,0). The initial color index is 1."
  - (2.8, p. 21) Added secondary color command for vertex arrays:

    Change first paragraph to read:

        "The vertex specification commands described in section 2.7 accept
        data in almost any format, but their use requires many command
        executions to specify even simple geometry. Vertex data may also be
        placed into arrays that are stored in the client's address space.
        Blocks of data in these arrays may then be used to specify multiple
        geometric primitives through the execution of a single GL command.
        The client may specify up to seven arrays: one each to store edge
        flags, texture coordinates, colors, secondary colors, color indices,
        normals, and vertices. The commands"

    Add to functions listed following first paragraph:

        void SecondaryColorPointerEXT(int size, enum type, sizei stride,
                                      void *pointer)

    Add to table 2.4 (p. 22):

        Command                     Sizes   Types
        -------                     -----   -----
        SecondaryColorPointerEXT    3       byte,ubyte,short,ushort,int,uint,
                                            float,double

    Starting with the second paragraph on p. 23, change to add
    SECONDARY_COLOR_ARRAY_EXT:

        "An individual array is enabled or disabled by calling one of

            void EnableClientState(enum array)
            void DisableClientState(enum array)

        with array set to EDGE_FLAG_ARRAY, TEXTURE_COORD_ARRAY, COLOR_ARRAY,
        SECONDARY_COLOR_ARRAY_EXT, INDEX_ARRAY, NORMAL_ARRAY, or
        VERTEX_ARRAY, for the edge flag, texture coordinate, color,
        secondary color, color index, normal, or vertex array, respectively.

        The ith element of every enabled array is transferred to the GL by
        calling

            void ArrayElement(int i)

        For each enabled array, it is as though the corresponding command
        from section 2.7 or section 2.6.2 were called with a pointer to
        element i. For the vertex array, the corresponding command is
        Vertex<size><type>v, where <size> is one of [2,3,4], and <type> is
        one of [s,i,f,d], corresponding to array types short, int, float,
        and double respectively. The corresponding commands for the edge
        flag, texture coordinate, color, secondary color, color index, and
        normal arrays are EdgeFlagv, TexCoord<size><type>v,
        Color<size><type>v, SecondaryColor3<type>vEXT, Index<type>v, and
        Normal<type>v, respectively..."
    Change pseudocode on p. 27 to disable secondary color array for
    canned interleaved array formats. After the lines

            DisableClientState(EDGE_FLAG_ARRAY);
            DisableClientState(INDEX_ARRAY);

        insert the line

            DisableClientState(SECONDARY_COLOR_ARRAY_EXT);

    Substitute "seven" for every occurence of "six" in the final paragraph
    on p. 27.

  - (2.12, p. 41) Add secondary color to the current rasterpos state.

    Change the last paragraph to read

        "The current raster position requires five single-precision
        floating-point values for its x_w, y_w, and z_w window coordinates,
        its w_c clip coordinate, and its eye coordinate distance, a single
        valid bit, a color (RGBA color, RGBA secondary color, and color
        index), and texture coordinates for associated data. In the initial
        state, the coordinates and texture coordinates are both $(0,0,0,1)$,
        the eye coordinate distance is 0, the valid bit is set, the
        associated RGBA color is $(1,1,1,1)$, the associated RGBA secondary
        color is $(0,0,0,0)$, and the associated color index color is 1. In
        RGBA mode, the associated color index always has its initial value;
        in color index mode, the RGBA color and and secondary color always
        maintain their initial values."

  - (2.13, p. 43) Change second paragraph to acknowledge two colors when
    lighting is disabled:

        "Next, lighting, if enabled, produces either a color index or
        primary and secondary colors. If lighting is disabled, the current
        color index or current color (primary color) and current secondary
        color are used in further processing. After lighting, RGBA colors
        are clamped..."

  - (Figure 2.8, p. 42) Change to show primary and secondary RGBA colors in
    both lit and unlit paths.

  - (2.13.1, p. 44) Change so that the second paragraph starts:

    "Lighting may be in one of two states:

     1. Lighting Off. In this state, the current color and current secondary
        color are assigned to the vertex primary color and vertex secondary
        color, respectively.

     2. ..."
```

- (2.13.1, p. 48) Change the sentence following equation 2.5 (for spot_i) so that color sum is implicitly enabled when SEPARATE_SPECULAR_COLOR is set:

  "All computations are carried out in eye coordinates. When c_es = SEPARATE_SPECULAR_COLOR, it is as if color sum (see section 3.9) were enabled, regardless of the value of COLOR_SUM_EXT."

- (3.9, p. 136) Change the first paragraph to read

  "After texturing, a fragment has two RGBA colors: a primary color c_pri (which texturing, if enabled, may have modified) and a secondary color c_sec.

  If color sum is enabled, the components of these two colors are summed to produce a single post-texturing RGBA color c (the A component of the secondary color is always 0). The components of c are then clamped to the range [0,1]. If color sum is disabled, then c_pri is assigned to the post texturing color. Color sum is enabled or disabled using the generic Enable and Disable commands, respectively, with the symbolic constant COLOR_SUM_EXT.

  The state required is a single bit indicating whether color sum is enabled or disabled. In the initial state, color sum is disabled."

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

None

Additions to the GLX Specification

None

GLX Protocol

Eight new GL rendering commands are added. The following commands are sent to the server as part of a glXRender request:

```
    SecondaryColor3bvEXT
        2           8               rendering command length
        2           4126            rendering command opcode
        1           INT8            v[0]
        1           INT8            v[1]
        1           INT8            v[2]
        1                           unused

    SecondaryColor3svEXT
        2           12              rendering command length
        2           4127            rendering command opcode
        2           INT16           v[0]
        2           INT16           v[1]
        2           INT16           v[2]
        2                           unused
        SecondaryColor3ivEXT
            2           16              rendering command length
            2           4128            rendering command opcode
            4           INT32           v[0]
            4           INT32           v[1]
            4           INT32           v[2]

        SecondaryColor3fvEXT
            2           16              rendering command length
            2           4129            rendering command opcode
            4           FLOAT32         v[0]
            4           FLOAT32         v[1]
            4           FLOAT32         v[2]

        SecondaryColor3dvEXT
            2           28              rendering command length
            2           4130            rendering command opcode
            8           FLOAT64         v[0]
            8           FLOAT64         v[1]
            8           FLOAT64         v[2]

        SecondaryColor3ubvEXT
            2           8               rendering command length
            2           4131            rendering command opcode
            1           CARD8           v[0]
            1           CARD8           v[1]
            1           CARD8           v[2]
            1                           unused

        SecondaryColor3usvEXT
            2           12               rendering command length
            2           4132            rendering command opcode
            2           CARD16          v[0]
            2           CARD16          v[1]
            2           CARD16          v[2]
            2                           unused

        SecondaryColor3uivEXT
            2           16               rendering command length
            2           4133            rendering command opcode
            4           CARD32          v[0]
            4           CARD32          v[1]
            4           CARD32          v[2]
```

Errors

```
    INVALID_VALUE is generated if SecondaryColorPointerEXT parameter <size>
    is not 3.

    INVALID_ENUM is generated if SecondaryColorPointerEXT parameter <type>
    is not BYTE, UNSIGNED_BYTE, SHORT, UNSIGNED_SHORT, INT, UNSIGNED_INT,
    FLOAT, or DOUBLE.

    INVALID_VALUE is generated if SecondaryColorPointerEXT parameter
    <stride> is negative.
```

New State

(table 6.5, p. 195)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
CURRENT_SECONDARY_COLOR_EXT	C	GetIntegerv, GetFloatv	(0,0,0,0)	Current secondary color	2.7	current

(table 6.6, p. 197)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
SECONDARY_COLOR_ARRAY_EXT	B	IsEnabled	False	Sec. color array enable	2.8	vertex-array
SECONDARY_COLOR_ARRAY_SIZE_EXT	Z+	GetIntegerv	3	Sec. colors per vertex	2.8	vertex-array
SECONDARY_COLOR_ARRAY_TYPE_EXT	Z8	GetIntegerv	FLOAT	Type of sec. color components	2.8	vertex-array
SECONDARY_COLOR_ARRAY_STRIDE_EXT	Z+	GetIntegerv	0	Stride between sec. colors	2.8	vertex-array
SECONDARY_COLOR_ARRAY_POINTER_EXT	Y	GetPointerv	0	Pointer to the sec. color array	2.8	vertex-array

(table 6.8, p. 198)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
COLOR_SUM_EXT	B	IsEnabled	False	True if color sum enabled	3.9	fog/enable

Name

EXT_separate_specular_color

Name Strings

GL_EXT_separate_specular_color

Version

$Date: 1997/10/05 00:16:23 $ $Revision: 1.3 $

Number

144

Dependencies

None

Overview

This extension adds a second color to rasterization when lighting is enabled. Its purpose is to produce textured objects with specular highlights which are the color of the lights. It applies only to rgba lighting.

The two colors are computed at the vertexes. They are both clamped, flat-shaded, clipped, and converted to fixed-point just like the current rgba color (see Figure 2.8). Rasterization interpolates both colors to fragments. If texture is enabled, the first (or primary) color is the input to the texture environment; the fragment color is the sum of the second color and the color resulting from texture application. If texture is not enabled, the fragment color is the sum of the two colors.

A new control to LightModel*, LIGHT_MODEL_COLOR_CONTROL_EXT, manages the values of the two colors. It takes values: SINGLE_COLOR_EXT, a compatibility mode, and SEPARATE_SPECULAR_COLOR_EXT, the object of this extension. In single color mode, the primary color is the current final color and the secondary color is 0.0. In separate specular mode, the primary color is the sum of the ambient, diffuse, and emissive terms of final color and the secondary color is the specular term.

There is much concern that this extension may not be compatible with the future direction of OpenGL with regards to better lighting and shading models. Until those impacts are resolved, serious consideration should be given before adding to the interface specified herein (for example, allowing the user to specify a second input color).

Issues

* Where is emissive included?

  RESOLVED - Emissive is included with the ambient and diffuse

```
    terms.  Grouping emissive with specular (the "proper" thing) could
    be implemented with a new value for the color control.

* Should there be two colors when not lighting or with index
  lighting?

    RESOLVED - The answer is probably yes--there should be two colors
    when lighting is disabled and there could be an incorporation of
    two colors with index lighting; but these are beyond the scope of
    this extension.  Further, attempts to accomplish these may not be
    compatible with the future direction of OpenGL with respect to
    high quality lighting and shading models.

  * What happens when texture is disabled?

    RESOLVED - The extension specifies to add the two colors when
    texture is disabled.  This is compatible with the philosophy of
    "if texture is disabled, this mode does not apply".
```

New Procedures and Functions

None.

New Tokens

Accepted by the <pname> parameter of LightModel*, and also by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

```
    LIGHT_MODEL_COLOR_CONTROL_EXT       0x81F8
```

Accepted by the <param> parameter of LightModel* when <pname> is LIGHT_MODEL_COLOR_CONTROL_EXT:

```
    SINGLE_COLOR_EXT                    0x81F9
    SEPARATE_SPECULAR_COLOR_EXT         0x81FA
```

Additions to Chapter 2 of the 1.0 Specification (OpenGL Operation)

```
  - (2.13, p. 40) Rework the second paragraph to acknowledge two
    colors:

    "Next, lighting, if enabled, produces either a color index or
    primary and secondary colors.  If lighting is disabled, the
    current color index or color is used in further processing (the
    current color is the primary color and the secondary color is 0).
    After lighting, colors are clamped..."

  - (Figure 2.8, p. 41) Change RGBA to primary RGBA and secondary RGB:

    Ideally, there might be an RGB2 underneath RGBA (both places).
    Alternatively, a note in the caption could clarify that RGBA
    referred to the primary RGBA and a secondary RGB.  (Speaking of
    the caption, the part about "m is the number of bits an R, G, B,
    or A component" could be removed as m doesn't appear in the
    diagram.)
```

```
- (2.13.1, p. 42) Rework the opening of this section to not imply a
  single color:

  In the first sentence, change "a color" to "colors".  Rephrase the
  itemization of the two lighting states to:

  "1. Lighting Off. In this state, the current color is assigned to
      the vertex primary color.  The vertex secondary color is 0.

   2. Lighting On.  In this state, the vertex primary and secondary
      colors are computed from the current lighting parameters."

- (Table 2.7, p.44) Add new entry (at the bottom):

  Parameter  Type  Default Value     Description
  ---------  ----  ----------------  ------------------------------
  c_es       enum  SINGLE_COLOR_EXT  controls computation of colors

- (p. 45, top of page) Rephrase the first line and equation:

  "Lighting produces two colors at a vertex: a primary color c_1 and
  a secondary color c_2.  The values of c_1 and c_2 depend on the
  light model color control, c_es (note: c_es should be in italics
  and c_1 and c_2 in bold, so this really won't be as confusing as
  it seems).  If c_es = SINGLE_COLOR_EXT, then the equations to
  compute c_1 and c_2 are (note: the equation for c_1 is the current
  equation for c):

    c_1 = e_cm
        + a_cm * a_cs
        + SUM(att_i * spot_i * (a_cm * a_cli
                               + dot(n, VP_pli) * d_cm * d_cli
                               + f_i * dot(n, h_i)^s_rm * s_cm * s_cli)
    c_2 = 0

  If c_es = SEPARATE_SPECULAR_COLOR_EXT, then:

    c_1 = e_cm
        + a_cm * a_cs
        + SUM (att_i * spot_i * (a_cm * a_cli
                                + (n dot VP_pli) * d_cm * d_cli)

    c_2 = SUM(att_i * spot_i * (f_i * (n dot h_i)^s_rm * s_cm * s_cli)

- (p. 45, second paragraph from bottom) Clarify that A is in the
  primary color:

  After the sentence "The value of A produced by lighting is the
  alpha value associated with d_cm", add "A is always associated
  with the primary color c_1; c_2 has no alpha component."

- (Table 2.8, p. 48) Add a new entry (at the bottom):

  Parameter  Name                          Number of values
  ---------  ----------------------------  ----------------
  c_es       LIGHT_MODEL_COLOR_CONTROL_EXT        1
```

- (2.13.6, p. 51) Clarify that both primary and secondary colors are clamped:

  Replace "RGBA" in the first line of the section with "both primary and secondary".

- (2.13.7, p. 52) Clarify what happens to primary and secondary colors when flat shading--reword the first paragraph:

  "A primitive may be flatshaded, meaning that all vertices of the primitive are assigned the same color index or primary and secondary colors. These come from the vertex that spawned the primitive. For a point, these are the colors associated with the point. For a line segment, they are the colors of the second (final) vertex of the segment. For a polygon, they come from a selected vertex depending on how the polygon was generated. Table 2.9 summarizes the possibilities."

- (2.13.8, p. 52) Rework to not imply a single color:

  In the second sentence, change "If the color is" to "Those" and ", it is" to "are". In the first sentence of the next paragraph, change "the color" to "two colors".

Additions to Chapter 3 of the 1.0 Specification (Rasterization)

- (Figure 3.1, p. 55) Add a box between texturing and fog called "color sum".

- (3.8, p. 85) In the first paragraph, second sentence, insert "primary" before RGBA. Insert after this sentence "Texturing does not affect the secondary color."

- (new section before 3.9) Insert new section titled "Color Sum":

  "At the beginning of this stage in RGBA mode, a fragment has two colors: a primary RGBA color (which texture, if enabled, may have modified) and a secondary RGB color. This stage sums the R, G, and B components of these two colors to produce a single RGBA color. If the resulting RGB values exceed 1.0, they are clamped to 1.0.

  In color index mode, a fragment only has a single color index and this stage does nothing."

Additions to Chapter 4 of the 1.0 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the 1.0 Specification (Special Functions)

- (5.3, p. 137) Specify that feedback returns the primary color by changing the last sentence of the large paragraph in the middle of the page to:

    "The colors returned are the primary colors. These colors and the
    texture coordinates are those resulting from the clipping operations
    as described in section 2.13.8."

Additions to Chapter 6 of the 1.0 Specification (State and State Requests)

  - (Table 6.9, p. 157) Add:

```
    Get Value - LIGHT_MODEL_COLOR_CONTROL_EXT
    Type - Z2
    Get Cmnd - GetIntegerv
    Initial Value - SINGLE_COLOR_EXT
    Description - color control
    Sec. - (whatever it ends up as)
    Attribute - lighting
```

Additions to the GLX Specification

  None.

GLX Protocol

  None.

Errors

  None.

New State

  (see changes to table 6.9)

Name

    EXT_shadow_funcs

Name Strings

    GL_EXT_shadow_funcs

Status

    Complete

Version

```
    Last Modified Date:  $Date: 2002/03/22 $
    NVIDIA Revision:  $Revision: #5 $
```

Number

    267

Dependencies

```
    OpenGL 1.1 is required.
    ARB_depth_texture is required.
    ARB_shadow is required.
    This extension is written against the OpenGL 1.3 Specification.
```

Overview

```
    This extension generalizes the GL_ARB_shadow extension to support all
    eight binary texture comparison functions rather than just GL_LEQUAL
    and GL_GEQUAL.
```

IP Status

    None.

Issues

```
    (1) What should this extension be called?

      RESOLUTION: EXT_shadow_funcs.  The extension adds new texture
      compare (shadow) comparison functions to ARB_shadow.

    (2) Are there issues with GL_EQUAL and GL_NOTEQUAL?

      The GL_EQUAL mode (and GL_NOTEQUAL) may be difficult to obtain
      well-defined behavior from. This is because there is no guarantee
      that the divide done by the shadow mapping r/q division is going
      to exactly match the z/w perspective divide and depth range scale
      & bias used to generate depth values.  Perhaps it can work in a
      well-defined manner in orthographic views or if you can guarantee
      that the texture hardware's r/q is computed with the same hardware
      used to compute z/w (NVIDIA's NV_texture_shader extension can
      provide such a guarantee).
```

Similiarly, GL_LESS and GL_GREATER or only different from GL_LEQUAL and GL_GEQUAL respectively by a single unit of depth precision which may make the difference between these modes very subtle.

New Procedures and Functions

None

New Tokens

None

Additions to Chapter 2 of the 1.3 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.3 Specification (Rasterization)

Section 3.8.4, Texture Parameters, p. 133, update table 3.19 with the following new legal values for TEXTURE_COMPARE_FUNC_ARB:

Name	Type	Legal Values
TEXTURE_COMPARE_FUNC_ARB	enum	LEQUAL, GEQUAL, LESS, GREATER, EQUAL, NOTEQUAL, ALWAYS, NEVER

After section 3.8.12, Texture Environments and Texture Functions, p. 149, update the texture compare pseudo-code in section 3.8.13.1 (as added by ARB_shadow):

```
if TEXTURE_COMPARE_MODE_ARB = NONE

    r = Dt

else if TEXTURE_COMPARE_MODE_ARB = COMPARE_R_TO_TEXTURE_ARB

    if TEXTURE_COMPARE_FUNC_ARB = LEQUAL

          { 1.0,  if R <= Dt
      r = {
          { 0.0,  if R > Dt

    else if TEXTURE_COMPARE_FUNC_ARB = GEQUAL

          { 1.0,  if R >= Dt
      r = {
          { 0.0,  if R < Dt

    else if TEXTURE_COMPARE_FUNC_ARB = LESS

          { 1.0,  if R < Dt
      r = {
          { 0.0,  if R >= Dt

    else if TEXTURE_COMPARE_FUNC_ARB = GREATER
```

```
                 { 1.0,  if R > Dt
             r = {
                 { 0.0,  if R <= Dt

            else if TEXTURE_COMPARE_FUNC_ARB = EQUAL

                 { 1.0,  if R == Dt
             r = {
                 { 0.0,  if R != Dt

            else if TEXTURE_COMPARE_FUNC_ARB = NOTEQUAL

                 { 1.0,  if R != Dt
             r = {
                 { 0.0,  if R == Dt

            else if TEXTURE_COMPARE_FUNC_ARB = ALWAYS

             r = 1.0

            else if TEXTURE_COMPARE_FUNC_ARB = NEVER

             r = 0.0

            endif

            if DEPTH_TEXTURE_MODE_ARB = LUMINANCE

                Lt = r

            else if DEPTH_TEXTURE_MODE_ARB = INTENSITY

                It = r

            else if DEPTH_TEXTURE_MODE_ARB = ALPHA

                At = r

            endif

        endif
```

Additions to Chapter 4 of the 1.3 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.3 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.3 Specification (State and State Requests)

None

Additions to the GLX Specification

None

Errors

INVALID_ENUM is generated if TexParameter[if][v] parameter <pname>
is TEXTURE_COMPARE_FUNC_ARB and parameter <param> is not one of
LEQUAL, GEQUAL, LESS, GREATER, EQUAL, NOTEQUAL, ALWAYS, or NEVER.

New State

In table 6.16, Texture Objects, p. 224, add the following:

Get Value	Type	Get Command	Initial Value	Description	Sec.	Attribute
TEXTURE_COMPARE_FUNC_ARB	Z_8	GetTexParameter[if]v	LEQUAL	compare func	3.8.13	texture

New Implementation Dependent State

None

Revision History

None

NV20 Implementation Details

NV20 (GeForce3 and Quadro DCC) will fallback to software rasterization
if two or more texture units have distinct TEXTURE_COMPARE_FUNC_ARB
settings that are not opposites (eg, GL_EQUAL and GL_NOTEQUAL).
This is not an issue on NV25 (GeForce4 and Quadro4).

Name

    EXT_shared_texture_palette

Name Strings

    GL_EXT_shared_texture_palette

Version

    $Date: 2004/03/24 23:23:04 $ $Revision: 1.4 $

Number

    141

Support

Mesa.

Selected NVIDIA GPUs: NV1x (GeForce 256, GeForce2, GeForce4 MX, GeForce4 Go, Quadro, Quadro2), NV2x (GeForce3, GeForce4 Ti, Quadro DCC, Quadro4 XGL), and NV3x (GeForce FX 5xxxx, Quadro FX 1000/2000/3000). NV3 (Riva 128) and NV4 (TNT, TNT2) GPUs and NV4x GPUs do NOT support this functionality (no hardware support). Future NVIDIA GPU designs will no longer support paletted textures.

S3 ProSavage, Savage 2000.

3Dfx Voodoo3, Voodoo5.

3Dlabs GLINT.

Dependencies

EXT_paletted_texture is required.

Overview

EXT_shared_texture_palette defines a shared texture palette which may be used in place of the texture object palettes provided by EXT_paletted_texture. This is useful for rapidly changing a palette common to many textures, rather than having to reload the new palette for each texture. The extension acts as a switch, causing all lookups that would normally be done on the texture's palette to instead use the shared palette.

Issues

* Do we want to use a new <target> to ColorTable to specify the shared palette, or can we just infer the new target from the corresponding Enable?

* A future extension of larger scope might define a "texture palette object" and bind these objects to texture objects dynamically, rather than making palettes part of the texture object state as the current EXT_paletted_texture spec does.

* Should there be separate shared palettes for 1D, 2D, and 3D textures?

  Probably not; palette lookups have nothing to do with the dimensionality of the texture. If multiple shared palettes are needed, we should define palette objects.

* There's no proxy mechanism for checking if a shared palette can be defined with the requested parameters. Will it suffice to assume that if a texture palette can be defined, so can a shared palette with the same parameters?

* The changes to the spec are based on changes already made for EXT_paletted_texture, which means that all three documents must be referred to. This is quite difficult to read.

* The changes to section 3.8.6, defining how shared palettes are enabled and disabled, might be better placed in section 3.8.1. However, the underlying EXT_paletted_texture does not appear to modify these sections to define exactly how palette lookups are done, and it's not clear where to put the changes.

* How does the shared texture palette interact with multitexture support? There is a single global shared texture palette that all texture units utilize (as opposed to a shared texture palette per texture unit).

New Procedures and Functions

None

New Tokens

Accepted by the <pname> parameters of GetBooleanv, GetIntegerv, GetFloatv, GetDoublev, IsEnabled, Enable, Disable, ColorTableEXT, ColorSubTableEXT, GetColorTableEXT, GetColorTableParameterivEXT, and GetColorTableParameterfd EXT:

```
SHARED_TEXTURE_PALETTE_EXT              0x81FB
```

Additions to Chapter 2 of the 1.1 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.1 Specification (Rasterization)

Section 3.8, 'Texturing,' subsection 'Texture Image Specification' is modified as follows:

In the Palette Specification Commands section, the sentence beginning 'target specifies which texture is to' should be changed to:

target specifies the texture palette or shared palette to be changed, and may be one of TEXTURE_1D, TEXTURE_2D, PROXY_TEXTURE_1D, PROXY_TEXTURE_2D, TEXTURE_3D_EXT, PROXY_TEXTURE_3D_EXT, or SHARED_TEXTURE_PALETTE_EXT.

In the 'Texture State and Proxy State' section, the sentence beginning 'A texture's palette is initially...' should be changed to:

There is also a shared palette not associated with any texture, which may override a texture palette. (Even when multiple texture units are available, there is still only a single shared texture palette.) All palettes are initially...

Section 3.8.6, 'Texture Application' is modified by appending the following:

Use of the shared texture palette is enabled or disabled using the generic Enable or Disable commands, respectively, with the symbolic constant SHARED_TEXTURE_PALETTE_EXT.

The required state is one bit indicating whether the shared palette is enabled or disabled. In the initial state, the shared palettes is disabled.

Additions to Chapter 4 of the 1.1 Specification (Per-Fragment Operations and the Frame buffer)

None

Additions to Chapter 5 of the 1.1 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.1 Specification (State and State Requests)

In the section on GetTexImage, the sentence beginning 'If format is not COLOR_INDEX...' should be changed to:

If format is not COLOR_INDEX, the texture's indices are passed through the texture's palette, or the shared palette if one is enabled, and the resulting components are assigned among R, G, B, and A according to Table 6.1.

In the GetColorTable section, the first sentence of the second paragraph should be changed to read:

GetColorTableEXT retrieves the texture palette or shared palette given by target.

The first sentence of the third paragraph should be changed to read:

Palette parameters can be retrieved using

```
void GetColorTableParameterivEXT(enum target, enum pname, int *params);
void GetColorTableParameterfvEXT(enum target, enum pname, float *params);
```

target specifies the texture palette or shared palette being queried and pname controls which parameter value is returned.

Additions to the GLX Specification

None

New State

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
SHARED_TEXTURE_PALETTE_EXT	B	IsEnabled	False	shared texture palette enable	3.8.6	texture/enable
SHARED_TEXTURE_PALETTE_EXT	I	GetColorTableEXT	empty	shared texture palette table	3.8	-
COLOR_TABLE_FORMAT_EXT	Zn	GetColorTableParameterivEXT	RGBA	shared texture palette format	3.8	-
COLOR_TABLE_WIDTH_EXT	Z+	GetColorTableParameteriv	0	shared texture palette width	3.8	-
COLOR_TABLE_x_SIZE_EXT	6xZ+	GetColorTableParameteriv	0	shared texture palette component sizes	3.8	-

New Implementation Dependent State

None

Revision History

July 10, 2002 - Added "New State" tables entries. Clarify that there is a single global shared texture palette, rather than a per-texture unit palette when multitexture is available.

March 24, 2004 - Document vendor support for this extension; note that future NVIDIA GPU designs will not support this extension.

Name

    EXT_stencil_clear_tag

Name Strings

    GL_EXT_stencil_clear_tag

Contact

Mark J. Kilgard, NVIDIA Corporation (mjk 'at' nvidia.com)

Notice

Status

Implemented, September 2004

Advertised and hardware-supported on NVIDIA GeForce 6 TurboCache GPUs.

Version

```
Last Modified:      10/15/2004
NVIDIA Revision:    4
```

Number

314

Dependencies

Written based on the wording of the OpenGL 1.5 specification.

Overview

Stencil-only framebuffer clears are increasingly common as 3D applications are now using rendering algorithms such as stenciled shadow volume rendering for multiple light sources in a single frame, recent "soft" stenciled shadow volume techniques, and stencil-based constructive solid geometry techniques. In such algorithms there are multiple stencil buffer clears for each depth buffer clear. Additionally in most cases, these algorithms do not require all of the 8 typical stencil bitplanes for their stencil requirements. In such cases, there is the potential for unused stencil bitplanes to encode a "stencil clear tag" in such a way to reduce the number of actual stencil clears. The idea is that switching to an unused stencil clear tag logically corresponds to when an application would otherwise perform a framebuffer-wide stencil clear.

This extension exposes an inexpensive hardware mechanism for amortizing the cost of multiple stencil-only clears by using a client-specified number of upper bits of the stencil buffer to maintain a per-pixel stencil tag.

The upper bits of each stencil value is treated as a tag that indicates the state of the upper bits of the "stencil clear tag" state when the stencil value was last written. If a stencil value is read and its upper bits containing its tag do NOT match the current upper bits of the stencil clear tag state, the stencil value is substituted with the lower bits of the stencil clear tag (the reset value).

Either way, the upper tag bits of the stencil value are ignored by subsequent stencil function and operation processing of the stencil value.

When a stencil value is written to the stencil buffer, its upper bits are overridden with the upper bits of the current stencil clear tag state so subsequent reads, prior to any subsequent stencil clear tag state change, properly return the updated lower bits.

In this way, the stencil clear tag functionality provides a way to replace multiple bandwidth-intensive stencil clears with very inexpensive update of the stencil clear tag state.

If used as expected with the client specifying 3 bits for the stencil tag, every 7 of 8 stencil-only clears of the entire stencil buffer can be substituted for an update of the current stencil clear tag rather than an actual update of all the framebuffer's stencil values. Still, every 8th clear must be an actual stencil clear. The net effect is that the aggregate cost of stencil clears is reduced by a factor of 1/(2^n) where n is the number of bits devoted to the stencil tag.

The application specifies two new pieces of state: 1) the number of upper stencil bits, n, assigned to maintain the tag bits for each stencil value within the stencil buffer, and 2) a stencil clear tag value that packs the current tag and a reset value into a single integer values. The upper n bits of the stencil clear tag value specify the current tag while the lower s-min(n,s) bits specify the current reset value, where s is the number of bitplanes in the stencil buffer and n is the current number of stencil tag bits.

If zero stencil clear tag bits are assigned to the stencil tag encoding, then the stencil buffer operates in the conventional manner.

Issues

1) *Can the stencil clear tag state be switched at anytime?*

RESOLUTION: Yes. The state controls the interpretation of the stencil values without actually change the values within the stencil buffer. So, for example, it is possible to render to the stencil buffer with 3 tag bits and then switch to 4 tag bits and a different reset value.

The effect of changing stencil clear tag state is well-defined though perhaps not useful.

The motivation for this decision is to make the underlying hardware implementation simple and not encumber operations such as stencil readback with extra expense to re-interpret stencil values.

2) *Can two distinct OpenGL rendering contexts render to the same framebuffer but with different stencil clear tag state?*

RESOLUTION: Yes. The stencil buffer contains raw stencil values whose interpretation and update may be different for the two contexts, but the values themselves are the same.

The motivation for this is that it avoids trying to coordinate two different contexts into maintaining the same interpretation of the stencil buffer. Different contexts can each view the stencil buffer values differently based on their own stencil clear tag state.

3) *For the purposes of the stencil comparison and stencil operations, how are upper bits of the read stencil value treated?*

RESOLUTION: The upper n bits where n is the current value of stencil tag bits (GL_STENCIL_TAG_BITS_EXT) are masked to zero when n is greater than zero.

For example, if a raw stencil value is 0xFA and the current stencil tag bits state is 3 with a stencil clear tag value of 0x82, the effective read stencil value is 0x02 because the upper 3 bits of 0xFA do not match the upper 3 bits of 0x82 and so the effective read stencil value is replaced with the lower 5 bits of 0x82 which is 0x02 while masking to zero the upper 3 bits. If instead, the stencil clear tag value was 0xEB, then the effective read stencil value is 0x1A because the upper 3 bits of 0xEB match the upper 3 bits of 0xFF so the effective read stencil value is 0xFA with the upper 3 bits masked to zero.

4) *How does the GL_INCR operation work when the stencil tag bits value is greater than zero?*

RESOLUTION: GL_INCR saturates to the value 2^(s-min(n,s))-1 where s is the number of stencil bits in the stencil buffer and n is the current value of stencil tag bits, rather than saturating to 2^s-1 or wrapping.

The motivation for this is to ensure that the stencil clear tag mechanism can fully emulate stencil buffers with fewer than s bits.

5) *What is the initial number of stencil tag bits?*

RESOLUTION: Zero. This is consistent with the conventional operation of the stencil buffer. The stencil clear tag value state is ignored when the stencil tag bits value is zero.

6) *Should glClear involving GL_STENCIL_BUFFER_BIT be subject to the stencil clear tag or tag bits state?*

RESOLUTION: No. An actual clear to the stencil buffer needs to reset the bitplanes allocated to the upper stencil tag bits as well as the lower bitplanes. So the stencil mask applies, but the stencil clear tag and tag bits state is ignored by glClear.

7) *Should glDrawPixels operations be subject to the stencil clear tag functionality?*

RESOLUTION: Yes. glDrawPixels to stencil already abides by the stencil write mask. Conceptually, think of glDrawPixels to stencil as being the GL_REPLACE operation where the value to be written comes from the glDrawPixels image rectangle rather than the stencil reference value.

The motivation is to allow the stencil clear tag mechanism to fully simulate a stencil buffer with fewer stencil bits.

If you want to write the entire stencil value, including upper bits that are allocated to encode the stencil tag, simply set the stencil tag bits state to zero for the duration of the glDrawPixels command.

8) *Should glReadPixels operations of type GL_STENCIL_INDEX be subject to the stencil clear tag state?*

RESOLUTION: Yes. So if you read stencil values from the stencil buffer, the n upper bits of each stencil value is compared to the n upper bits of the stencil clear tag value and if they mismatch, the lower s-min(n,s) bits of the stencil clear tag value (the reset value) are returned instead, where s is the number of stencil bitplanes and n is the current stencil tag bits value. In any case, the upper n bits of the stencil value are zeroed.

The motivation is to allow the stencil clear tag mechanism to fully simulate a stencil buffer with fewer stencil bits.

If you want to read the entire stencil value, including upper bits that are allocated to encode the stencil tag, then set the stencil tag bits state to zero for the duration of the glReadPixels command.

9) *Should glCopyPixels operations of type GL_STENCIL_INDEX be subject to the stencil clear tag state?*

RESOLUTION: Yes, because glReadPixels and glDrawPixels are both affected and glCopyPixels is defined in terms of glReadPixels and glDrawPixels.

10) *Should the current tag and reset value in the current stencil clear tag be packed into a single value where the stencil tag bits value divides the upper tag value bits from the lower reset value bits?*

RESOLUTION: Yes. This makes a lot of sense because there are always s bits required where n bits are for the current tag value and s-min(n,s) bits are for the reset value, where s is the number of stencil bitplanes and n is the number of stencil tag bits.

This packing also makes the explanation of how bit comparisons and the required masking operations operate in the specification language. It also naturally corresponds to how a hardware implementation would maintain the state.

11) *Clears can be scissored to only update a subrectangle of the entire framebuffer. Can the stencil clear tag facility accelerate scissored clears that do not clear the entire framebuffer?*

RESOLUTION: No. The stencil clear tag state is a single per-context state value that applies to the entire framebuffer.

For scissored clears to sufficiently small enough subrectangles of the screen, it may be more advantageous to perform an actual scissored clear if changing the current stencil clear tag value would be better used to save an subsequent actual stencil clear of the entire (or nearly the entire) framebuffer.

Doom 3 uses scissored clears when performing per-light stencil clears for its stenciled shadow volumes where the scissor is a 2D bound for the light's illumination.

12) *How does this extension interact with EXT_stencil_two_side or other two-sided stencil testing functionality such as that provided by OpenGL 2.0?*

RESOLUTION: The stencil clear tag state is not two-sided because it reflects the manner that stencil values in the stencil buffer are read to and written from the buffer rather than anything to do with the facingness of primitives.

13) *How does the GL_KEEP operation operate when the value of GL_STENCIL_TAG_BITS_EXT is greater than zero?*

RESOLUTION: GL_KEEP means no stencil write is performed so the pixel's stencil value is completely unchanged. This means the pixel's stencil value will still have the old stencil tag.

The rationale for this is that GL_KEEP will always avoid memory writes to the stencil buffer, even when the current stencil tag state does not match the tag of pixel's stencil value.

All other stencil operations must actually write the stencil tag bits into the upper bits of the pixel's stencil value if the old value's tag does not match the current stencil tag state. For example, if the value of GL_STENCIL_TAG_BITS_EXT is 3, the value of GL_STENCIL_CLEAR_TAG_EXT is 0x80, the stencil write mask is 0xFF, and a pixel's stencil value is 0x00, the result of a GL_ZERO stencil operation for this pixel is to write 0x80. into the stencil buffer.

14) *How does a stencil write mask of zero operate when the value of GL_STENCIL_GENERATION_BITS_EXT is greater than zero?*

RESOLUTION: A stencil write mask of zero means no stencil write is performed so the pixel's stencil value is completely unchanged. This means the pixel's stencil value will still have the old stencil tag bits.

The rationale for this is essentially the same for GL_KEEP's behavior in the previous issue.

15) *Conceptually, how does the stencil clear tag functionality augment the existing stencil processing pipeline?*

RESOLUTION: Unextended OpenGL stencil processing (ignoring the depth test interactions) says:

```
  read stencil value
    |
    v
  evaluate stencil function
    |
    v
  apply appropriate stencil operation
    |
    v
  if operation is non-GL_KEEP, write stencil value
```

The EXT_stencil_clear_tag functionality augments this pipeline with two new stages:

```
  read stencil value
    |
    v
  perform stencil clear tag "read merge"
    |
    v
  evaluate stencil function
    |
    v
  apply appropriate stencil operation
    |
    v
  perform stencil clear tag "write merge"
    |
    v
  if a non-KEEP operation, write stencil value
```

The new stencil clear tag merge stages are pass-through operations if the value of GL_STENCIL_TAG_BITS_EXT is zero (the initial state).

16) *Can you provide an example of how this stencil clear tag mechanism could be used to eliminate stencil clears for a stenciled shadow volume application with multiple light sources per frame.*

First assume the application's shadow complexity is such that scenes never exceed a shadow complexity of 31 (or 63 or 127) at any pixel, meaning a 5 (or 6 or 7) bit stencil buffer is sufficient to avoid artifacts.

The code assumes "Z fail" shadow volume rendering with two-sided stencil testing and an 8-bit stencil buffer.

So initialize the stencil-related state as follows:

```
  const GLint stencilTagBits = 3;  // or 2 or 1
  const int hasStencilClearTagExtension =
      queryExtension("GL_EXT_stencil_clear_tag");

  GLint stencilBits;
  GLuint maxStencilValue;
  GLint tagInit;
  GLint tagDecrement;
  GLint stencilClearTag;

  if (hasStencilClearTagExtension) {
      glGetIntegerv(GL_STENCIL_BITS, &stencilBits);
      maxStencilValue = (1U<<stencilBits)-1;
      assert(stencilBits > stencilTagBits);
      tagDecrement = 1<<(stencilBits - stencilTagBits);
      tagInit = ~(tagDecrement-1) & maxStencilValue;

      glStencilClearTagEXT(stencilTagBits, tagInit);
      glStencilClear(tagInit);
  } else {
      glStencilClear(0);
  }

  glEnable(GL_STENCIL_TWO_SIDE_EXT);
  glActiveStencilFaceEXT(GL_BACK);
  glStencilMask(~0);
  glActiveStencilFaceEXT(GL_FRONT);
  glStencilMask(~0);
```

Then rendering one frame of a shadowed scene looks like:

```
  int i;

  glDepthMask(1);
  glColorMask(1,1,1,1);

  if (hasStencilClearTagExtension) {
      stencilClearTag = tagInit;
      glStencilClearTagEXT(stencilTagBits, stencilClearTag);
  }
  glClear(GL_STENCIL_BUFFER_BIT |
          GL_DEPTH_BUFFER_BIT |
          GL_COLOR_BUFFER_BIT);

  glDisable(GL_BLEND);
  glDisable(GL_STENCIL_TEST);
  glDepthFunc(GL_LESS);
  glEnable(GL_DEPTH_TEST);

  renderDepthAndAmbient();

  glEnable(GL_BLEND);
  glBlendFunc(GL_ONE, GL_ONE);
  glEnable(GL_STENCIL_TEST);
  glDepthMask(0);
  glDepthFunc(GL_EQUAL);
```

```
      for (i=0; i<numberOfLights; i++) {
          if (i == 0) {
              // First light can hitches ride on frame's initial gang clear
          } else {
              // Subsequent lights must effect a clear
              if (hasStencilClearTagExtension) {
                  // Did start on a new set of tags?
                  if (stencilClearTag == tagInit) {
                      // If so, do real stencil clear and reset stencilClearTag.
                      glClear(GL_STENCIL_BUFFER_BIT);
                      // Decrement to next tag.
                      stencilClearTag -= tagDecrement;
                  }
                  // Are we out of tags?
                  else if (stencilClearTag == 0) {
                      // Reset to the initial tag.
                      stencilClearTag = tagInit;
                  } else {
                     // Decrement to next tag.
                     stencilClearTag -= tagDecrement;
                  }
                  glStencilClearTagEXT(stencilTagBits, stencilClearTag);
              } else {
                  // Actual per-light clear needed
                  glClear(GL_STENCIL_BUFFER_BIT);
              }
          }
          glActiveStencilFaceEXT(GL_BACK);
              glStencilFunc(GL_ALWAYS, 0, ~0);
              glStencilOp(GL_KEEP, GL_INCR_WRAP, GL_KEEP);
          glActiveStencilFaceEXT(GL_FRONT);
              glStencilFunc(GL_ALWAYS, 0, ~0);
              glStencilOp(GL_KEEP, GL_DECR_WRAP, GL_KEEP);
          glColorMask(0,0,0,0);

          renderShadowVolumesForLight(i);

          glActiveStencilFaceEXT(GL_BACK);
              glStencilFunc(GL_EQUAL, 0, ~0);
              glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
          glActiveStencilFaceEXT(GL_FRONT);
              glStencilFunc(GL_EQUAL, 0, ~0);
              glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
          glColorMask(1,1,1,1);

          renderLightingContributionForLight(i);
      }
```

A smarter implementation could include computation of the scissor (and depth bounds) for each light source. If the number of lights exceeds the number of available stencil tags, the lights with the smallest scissor area could be performed as actual scissored clears so the clears to the largest regions could be done as stencil clear tag state updates.

stencilTagBits can be adjusted based on the number of active lights. For example, if there are only 4 lights active, stencilTagBits could be 2 instead of 3 and thereby recover a bit of stencil precision for the shadow volume count.

17) *Why "s-min(n,s)" instead of simply "s-n" where s is the number of stencil bits and n is the number of stencil tag bits?*

RESOLVED: This makes sure if a context migrates to a drawable with fewer stencil bits than a drawable had when glStencilClearTagEXT was last called, the effect should be well-defined.

For example, if glStencilClearTagEXT(3,0) is called with an 8-bit stencil buffer and then that context is bound to a drawable with no stencil buffer (effectively, 0 bits), s-min(n,s) is zero rather than s-n being -3.

18) *Should the stencil reference value be ANDed with 2^(s-min(n,s))-1?*

RESOLOVED: Yes. this way the reference value and the compared stencil value compare a matching number of bits.

New Procedures and Functions

```
    StencilClearTagEXT(sizei stencilTagBits,
                       uint stencilClearTag)
```

New Tokens

Accepted by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

```
        STENCIL_TAG_BITS_EXT                0x88F2
        STENCIL_CLEAR_TAG_VALUE_EXT         0x88F3
```

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

None

Additions to Chapter 3 of the GL Specification (Rasterization)

None

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

Section 4.1.5 "Stencil Test" (page 174), add after the 1st paragraph:

"The command

```
       void StencilClearTagEXT(sizei stencilTagBits,
                               uint stencilClearTag);
```

controls the stencil clear tag state. stencilTagBits is a count of the number of most-significant stencil buffer bits involved in the stencil clear tag update. The error INVALID_VALUE is generated if stencilTagBits is negative or greater or equal to s."

Add after the 2nd sentence in the 2nd paragraph:

"The effective reference value used for the stencil comparison is ref ANDed with 2^(s-min(n,s))-1, where n is equal to stencilTagBits."

```
    Addd after the 2nd paragraph:

    "The stored stencil value used for the stencil comparison and
    subsequent stencil operations is obtained by reading the pixel's
    corresponding stencil value from the stencil buffer and possibly
    modifying that value based on the stencil clear tag state.

    The stored stencil value is modified prior to the stencil comparison
    if n (again where n is equal to stencilTagBits) is greater than zero;
    otherwise if zero, the stored stencil value remains unmodified.
    If n is greater than zero and the n most-significant bits of
    the stored stencil value all match the corresponding bits of
    the stencilClearTag, then the stored stencil value is ANDed with
    2^(s-min(n,s))-1. If n is greater than zero and the n most-significant
    bits of the stored stencil value do NOT match all the corresponding
    bits of the stencilClearTag, then the stored stencil value becomes
    stencilClearTag ANDed with 2^(s-min(n,s))-1. "

    Change the KEEP operation description in the 4th sentence to indicate
    that KEEP does not perform the stencil clear tag write merge:

    "keeping the current value without writing the stencil buffer,"

    Change the second sentence of the fourth paragraph to read:

    "Incrementing or decrementing with saturation clamps the stencil
    value at 0 and 2^(s-min(n,s))-1 so when stencilTagBits is zero the
    maximum saturation value is the maximum representable stencil value."

    Section 4.2.5 "Fine Control of Buffer Updates" (page 185), prior to
    the paragraph describing the StencilMask command, add:

    "Writes to the stencil buffer are controlled through a combination
    of stencil mask and stencil clear tag state."

    Then add after the paragraph describing the StencilMask command:

    "If the stencil mask ANDed with s^2(s-min(n,s))-1 is zero, no write
    occurs.  Otherwise, the pixel's stencil value is written with the
    value determined by the following C-style bit-wise expression:

       ( stencilClearTag & ~tagMask         ) |
       ( newValue        &  mask &  tagMask ) |
       ( storedValue     & ~mask &  tagMask )

    where tagMask is 2^(s-min(n,s))-1, n is the value of the
    stencil tag bits state, newValue is the stencil value to
    be written (after the stored value's potential modification due to
    stencil clear tag state AND after the effect of applying a stencil
    operation to the value), and storedValue is the pixel's stored
    stencil value after to its potential modification due to stencil
    clear tag state BUT BEFORE to any stencil operation that may have
    been performed (as discussed in section 4.1.5).  When n is zero,
    this is equivalent to

       ( newValue    &  mask ) |
       ( storedValue & ~mask )

    "
```

Section 4.2.3 "Clearing the Buffers", change the ClearStencil sentence to read:

"Similarly,

```
    void ClearStencil(int s);
```

takes a single integer argument that is the value to which to clear the stencil buffer. s is masked to the number of bitplanes in the stencil buffer. Clearing stencil ignores the stencil clear tag state."

Section 4.3.1 "Writing to the Stencil Buffer", change the last sentence to say:

"Finally, each stencil index is written to its indicated location in the framebuffer, subject to the current setting of StencilMask and StencilClearTagEXT (see section 4.2.5). This means the most-significant n stencil bitplanes cannot be written by DrawPixels where n is the current number of stencil tag bits."

Section 4.3.2 "Reading Pixels - Obtaining Pixels from the Framebuffer", change third paragraph to read:

"If the format is STENCIL_INDEX, then values are taken from the stencil buffer; again, if there is no stencil buffer, the error INVALID_OPERATION occurs. If the current stencil tag bits state is zero (see section 4.2.5), the read stencil value is unmodified when read. If the current stencil tag bits state is greater than zero, then the upper most-significant n bits of the read stencil value are compared to the corresponding n bits of the stencil clear tag value, where n is the current number of stencil tag bits. If these upper bits mismatch, the read stencil value is replaced with the lower s-min(n,s) bits of the stencil clear tag state (zeroing the upper n bits), where s is the number of stencil bitplanes. If the upper bits match, the upper n bits of the read stencil value are zeroed."

Additions to Chapter 6 of the GL Specification (State and State Requests)

None

Additions to the GLX Specification

None

GLX Protocol

A new GL rendering command is added. The following command is sent to the server as part of a glXRender request:

StencilClearTagEXT

```
        2           12              rendering command length
        2           4223            rendering command opcode
        4           INT32           stencilTagBits
        4           CARD32          stencilClearTag
```

Errors

INVALID_VALUE is generated by StencilClearTagEXT if stencilTagBits is negative or greater or equal to s where s is the number of bits in the stencil buffer.

New State

(table 6.19, page 245)

Get Value	Type	Get Command	Initial Value	Sec	Attribute
STENCIL_TAG_BITS_EXT	Z+	GetIntegerv	0	4.1.5	stencil-buffer
STENCIL_CLEAR_TAG_EXT	Z+	GetIntegerv	0	4.1.5	stencil-buffer

New Implementation Dependent State

None

Name

    EXT_stencil_two_side

Name Strings

    GL_EXT_stencil_two_side

Notice

Status

    Implemented in CineFX (NV30) Emulation driver, August 2002.
    Shipping in Release 40 NVIDIA driver for CineFX hardware, January 2003.

Version

    Last Modified Date: $Date: 2003/01/08 $
    $Id: //sw/main/docs/OpenGL/specs/GL_EXT_stencil_two_side.txt#6 $

Number

    268

Dependencies

    Written based on the OpenGL 1.3 specification.

    NV_packed_depth_stencil affects the definition of this extension.

Overview

This extension provides two-sided stencil testing where the stencil-related state (stencil operations, reference value, compare mask, and write mask) may be different for front- and back-facing polygons. Two-sided stencil testing may improve the performance of stenciled shadow volume and Constructive Solid Geometry (CSG) rendering algorithms.

Issues

Is this sufficient for shadow volume stencil update in a single pass?

RESOLUTION: Yes.

An application that wishes to increment the stencil value for rasterized depth-test passing fragments of front-facing polygons and decrement the stencil value for rasterized fragments of depth-test passing back-facing polygons in a single pass can use the following configuration:

```
  glDepthMask(0);
  glColorMask(0,0,0,0);
  glDisable(GL_CULL_FACE);
  glEnable(GL_STENCIL_TEST);
  glEnable(GL_STENCIL_TEST_TWO_SIDE_EXT);

  glActiveStencilFaceEXT(GL_BACK);
  glStencilOp(GL_KEEP,            // stencil test fail
              GL_KEEP,            // depth test fail
              GL_DECR_WRAP_EXT);  // depth test pass
  glStencilMask(~0);
  glStencilFunc(GL_ALWAYS, 0, ~0);

  glActiveStencilFaceEXT(GL_FRONT);
  glStencilOp(GL_KEEP,            // stencil test fail
              GL_KEEP,            // depth test fail
              GL_INCR_WRAP_EXT);  // depth test pass
  glStencilMask(~0);
  glStencilFunc(GL_ALWAYS, 0, ~0);

  renderShadowVolumePolygons();
```

Notice the use of EXT_stencil_wrap to avoid saturating decrements losing the shadow volume count. An alternative, using the conventional GL_INCR and GL_DECR operations, is to clear the stencil buffer to one half the stencil buffer value range, say 128 for an 8-bit stencil buffer. In the case, a pixel is "in shadow" if the final stencil value is greater than 128 and "out of shadow" if the final stencil value is 128. This does still create a potential for stencil value overflow if the stencil value saturates due to an increment or decrement. However saturation is less likely with two-sided stencil testing than the conventional two-pass approach because front- and back-facing polygons are mixed together, rather than processing batches of front-facing then back-facing polygons.

Contrast the two-sided stencil testing approach with the more or less equivalent approach using facingness-independent stencil testing:

```
  glDepthMask(0);
  glColorMask(0,0,0,0);
  glEnable(GL_CULL_FACE);
  glEnable(GL_STENCIL_TEST);

  glStencilMask(~0);
  glStencilFunc(GL_ALWAYS, 0, ~0);

  // Increment for front faces
  glCullFace(GL_BACK);
  glStencilOp(GL_KEEP,   // stencil test fail
              GL_KEEP,   // depth test fail
              GL_INCR);  // depth test pass

  renderShadowVolumePolygons();

  // Decrement for back faces
  glCullFace(GL_FRONT);
  glStencilOp(GL_KEEP,   // stencil test fail
              GL_KEEP,   // depth test fail
              GL_DECR);  // depth test pass

  renderShadowVolumePolygons();
```

Notice that all the render work implicit in renderShadowVolumePolygons is performed twice with the conventional approach, but only once with the two-sided stencil testing approach.

Should there be just front and back stencil test state, or should the stencil write mask also have a front and back state?

RESOLUTION: Both the stencil test and stencil write mask state should have front and back versions.

The shadow volume application for two-sided stencil testing does not require differing front and back versions of the stencil write mask, but we anticipate other applications where front and back write masks may be useful.

For example, it may be useful to draw a convex polyhedra such that (assuming the stencil bufer is cleared to the binary value 1010):

1) front-facing polygons that pass the depth test set stencil bit 0

2) front-facing polygons that fail the depth test zero stencil bit 1

3) back-facing polygons that pass the depth test set stencil bit 2

4) back-facing polygons that fail the depth test zero stencil bit 3

This could be accomplished in a single rendering pass using:

```
  glStencilMask(~0);
  glStencilClear(0xA);
  glClear(GL_STENCIL_BUFFER_BIT);

  glDepthMask(0);
  glColorMask(0,0,0,0);
  glDisable(GL_CULL_FACE);
  glEnable(GL_STENCIL_TEST);
  glEnable(GL_STENCIL_TEST_TWO_SIDE_EXT);

  glActiveStencilFaceEXT(GL_BACK);
  glStencilOp(GL_KEEP,      // stencil test fail
              GL_ZERO,      // depth test fail
              GL_REPLACE);  // depth test pass
  glStencilMask(0xC);
  glStencilFunc(GL_ALWAYS, 0x4, ~0);

  glActiveStencilFaceEXT(GL_FRONT);
  glStencilOp(GL_KEEP,      // stencil test fail
              GL_ZERO,      // depth test fail
              GL_REPLACE);  // depth test pass
  glStencilMask(0x3);
  glStencilFunc(GL_ALWAYS, 0x1, ~0);

  renderConvexPolyhedra();
```

Is there a performance advantage to using two-sided stencil testing?

RESOLUTION: It depends.

In a fill-rate limited situation, rendering front-facing primitives, then back-facing primitives in two passes will generate the same number of rasterized fragments as rendering front- and back-facing primitives in a single pass.

However, in other situations that are CPU-limited, transform-limited, or setup-limited, two-sided stencil testing can be faster than the conventional two-pass face culling rendering approaches. For example, if a lengthy vertex program is executed for every shadow volume vertex, rendering the shadow volume with a single two-sided stencil testing pass is advantageous.

Often applications using stencil shadow volume techniques require substantial CPU resources to determine potential silhouette boundaries to project shadow volumes from. If the shadow volume geometry generated by the CPU is only required to be sent to the GL once per-frame (rather than twice with the conventional technique), that can ease the CPU burden required to implement stenciled shadow volumes.

Should GL_FRONT_AND_BACK be accepted by glActiveStencilFaceEXT?

RESOLUTION: No.

GL_FRONT_AND_BACK is useful when materials are being updated for two-sided lighting because the front and back material are often identical and may change frequently (glMaterial calls are allowed within glBegin/glEnd pairs).

Two-sided stencil has no similiar performance justification.

It is also likely that forcing implementations to support this mode would increase the amount of overhead required to set stencil state, even for applications that don't use two-sided stencil.

How should the two-sided stencil enable operate?

RESOLUTION: It should be modeled after the way two-sided lighting works. There is a GL_LIGHTING enable and then an additional two-sided lighting mode. Unlike two-sided lighting which is a light model boolean, the two-sided stencil testing is a standard enable named GL_STENCIL_TEST_TWO_SIDE_EXT.

Here is the pseudo-code for the stencil testing enables:

```
if (glIsEnabled(GL_STENCIL_TEST)) {
  if (glIsEnabled(GL_STENCIL_TEST_TWO_SIDE_EXT) && primitiveType == polygon) {
    use two-sided stencil testing
  } else {
    use conventional stencil testing
  }
} else {
  no stencil testing
}
```

How should the two-sided stencil interact with glPolygonMode?

RESOLUTION: Primitive type is determined by the begin mode so GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_QUAD_STRIP, GL_QUADS, GL_TRIANGLE_FAN, and GL_POLYGON generate polygon primitives. If the polygon mode is set such that lines or points are rasterized, two-sided stencil testing still operates based on the original polygon facingness if stencil testing and two-sided stencil testing are enabled.

This is consistent with how two-sided lighting and face culling interact with glPolygonMode.

New Procedures and Functions

```
void ActiveStencilFaceEXT(enum face);
```

New Tokens

    Accepted by the <cap> parameter of Enable, Disable, and IsEnabled,
    and by the <pname> parameter of GetBooleanv, GetIntegerv,
    GetFloatv, and GetDoublev:

        STENCIL_TEST_TWO_SIDE_EXT   0x8910

    Accepted by the <face> parameter of ActiveStencilFaceEXT:

        FRONT
        BACK

    Accepted by the <pname> parameters of GetBooleanv, GetIntegerv,
    GetFloatv, and GetDoublev:

        ACTIVE_STENCIL_FACE_EXT   0x8911

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

    None

Additions to Chapter 3 of the GL Specification (Rasterization)

    None

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

 -- Section 4.1.5 "Stencil test"

    Replace the first paragraph in the section with:

    "The stencil test conditionally discards a fragment based on the
    outcome of a comparison between the value in the stencil buffer at
    location (xw,yw) and a reference value.

    The test is enabled or disabled with the Enable and Disable commands,
    using the symbolic constant STENCIL_TEST. When disabled, the stencil
    test and associated modifications are not made, and the fragment is
    always passed.

    Stencil testing may operate in a two-sided mode. Two-sided stencil
    testing is enabled or disabled with the Enable and Disable commands,
    using the symbolic constant STENCIL_TEST_TWO_SIDE_EXT. When stencil
    testing is disabled, the state of two-sided stencil testing does
    not affect fragment processing.

    There are two sets of stencil-related state, the front stencil
    state set and the back stencil state set. When two-sided stencil
    testing is enabled, stencil tests and writes use the front set of
    stencil state when processing fragments rasterized from non-polygon
    primitives (points, lines, bitmaps, image rectangles) and front-facing
    polygon primitives while the back set of stencil state is used when
    processing fragments rasterized from back-facing polygon primitives.
    For the purposes of two-sided stencil testing, a primitive is still
    considered a polygon even if the polygon is to be rasterized as

points or lines due to the current polygon mode. Whether a polygon is front- or back-facing is determined in the same manner used for two-sided lighting and face culling (see sections 2.13.1 and 3.5.1). When two-sided stencil testing is disabled, the front set of stencil state is always used when stencil testing fragments.

The active stencil face determines whether stencil-related commands update the front or back stencil state. The active stencil face is set with:

```
  void ActiveStencilFace(enum face);
```

where face is either FRONT or BACK. Stencil commands (StencilFunc, StencilOp, and StencilMask) that update the stencil state update the front stencil state if the active stencil face is FRONT and the back stencil state if the active stencil face is BACK. Additionally, queries of stencil state return the front or back stencil state depending on the current active stencil face.

The stencil test state is controlled with

```
   void StencilFunc(enum func, int ref, uint mask);
   void StencilOp(enum sfail, enum dpfail, enum dppass);"
```

Replace the third and second to the last sentence in the last paragraph in section 4.1.5 with:

"In the initial state, stencil testing and two-sided stencil testing are both disabled, the front and back stencil reference values are both zero, the front and back stencil comparison functions are ALWAYS, and the front and back stencil mask are both all ones. Initially, both the three front and the three back stencil operations are KEEP."

-- Section 4.2.2 "Fine Control of Buffer Updates"

Replace the last sentence of the third paragraph with:

"The initial state is for both the front and back stencil plane mask to be all ones. The clear operation always uses the front stencil write mask when clearing the stencil buffer."

-- Section 4.3.1 "Writing to the Stencil Buffer or to the Depth and Stencil Buffers"

Replace the final sentence in the first paragraph with:

"Finally, each stencil index is written to its indicated location in the framebuffer, subject to the current front stencil mask state (set with StencilMask), and if a depth component is present, if the setting of DepthMask is not FALSE, it is also written to the framebuffer; the setting of DepthTest is ignored."

Additions to Chapter 5 of the GL Specification (Special Functions)

None

Additions to Chapter 6 of the GL Specification (State and State Requests)

None

Additions to the GLX, WGL, and AGL Specification

None

GLX Protocol

A new GL rendering command is added. The following command is sent to the server as part of a glXRender request:

ActiveStencilFaceEXT

2	8	rendering command length
2	4220	rendering command opcode
4	ENUM	face

Errors

None

New State

(table 6.15, page 205) amend the following entries:

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
STENCIL_FUNC	2xZ8	GetIntegerv	ALWAYS	Stencil function	4.1.4	stencil-buffer
STENCIL_VALUE_MASK	2xZ+	GetIntegerv	1's	Stencil mask	4.1.4	stencil-buffer
STENCIL_REF	2xZ+	GetIntegerv	0	Stencil reference value	4.1.4	stencil-buffer
STENCIL_FAIL	2xZ6	GetIntegerv	KEEP	Stencil fail action	4.1.4	stencil-buffer
STENCIL_PASS_DEPTH_FAIL	2xZ6	GetIntegerv	KEEP	Stencil depth buffer fail action	4.1.4	stencil-buffer
STENCIL_PASS_DEPTH_PASS	2xZ6	GetIntegerv	KEEP	Stencil depth buffer pass action	4.1.4	stencil-buffer

[Type field is amended with "2x" prefix.]

(table 6.15, page 205) add the following entries:

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
STENCIL_TEST_TWO_SIDE_EXT	B	IsEnabled	False	Two-sided stencil test enable	4.1.4	stencil-buffer/enable
ACTIVE_STENCIL_FACE_EXT	Z2	GetIntegerv	FRONT	Active stencil face selector	4.1.4	stencil-buffer

(table 6.16, page 205) ammend the following entry:

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
STENCIL_WRITE_MASK	2xZ+	GetIntegerv	1's	Stencil buffer writemask	4.2.2	stencil-buffer

[Type field is amended with "2x" prefix.]

Revision History

    None

Name

    EXT_stencil_wrap

Name Strings

    GL_EXT_stencil_wrap

Version

    Date: 4/4/2002 Version 1.2

Number

    176

Dependencies

    None

Overview

Various algorithms use the stencil buffer to "count" the number of surfaces that a ray passes through. As the ray passes into an object, the stencil buffer is incremented. As the ray passes out of an object, the stencil buffer is decremented.

GL requires that the stencil increment operation clamps to its maximum value. For algorithms that depend on the difference between the sum of the increments and the sum of the decrements, clamping causes an erroneous result.

This extension provides an enable for both maximum and minimum wrapping of stencil values. Instead, the stencil value wraps in both directions.

Two additional stencil operations are specified. These new operations are similiar to the existing INCR and DECR operations, but they wrap their result instead of saturating it. This functionality matches the new stencil operations introduced by DirectX 6.

New Procedures and Functions

    None

New Tokens

Accepted by the <sfail>, <dpfail>, and <dppass> parameter of StencilOp:

```
    INCR_WRAP_EXT              0x8507
    DECR_WRAP_EXT              0x8508
```

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

    None

Additions to Chapter 3 of the GL Specification (Rasterization)

None

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

Section 4.1.4 "Stencil Test" (page 144), change the 3rd paragraph to read:

"... The symbolic constants are KEEP, ZERO, REPLACE, INCR, DECR, INVERT, INCR_WRAP_EXT, and DECR_WRAP_EXT. The correspond to keeping the current value, setting it to zero, replacing it with the reference value, incrementing it with saturation, decrementing it with saturation, bitwise inverting it, incrementing it without saturation, and decrementing it without saturation. For purposes of incrementing and decrementing, the stencil bits are considered as an unsigned integer. Incrementing or decrementing with saturation will clamp values at 0 and the maximum representable value. Incrementing or decrementing without saturation will wrap such that incrementing the maximum representable value results in 0 and decrementing 0 results in the maximum representable value. ..."

Additions to Chapter 5 of the GL Specification (Special Functions)

None

Additions to Chapter 6 of the GL Specification (State and State Requests)

None

Additions to the GLX Specification

None

GLX Protocol

None

Errors

INVALID_ENUM is generated by StencilOp if any of its parameters are not KEEP, ZERO, REPLACE, INCR, DECR, INVERT, INCR_WRAP_EXT, or DECR_WRAP_EXT.

New State

(table 6.15, page 205)

Get Value	Type	Get Command	Initial Value	Sec	Attribute
STENCIL_FAIL	Z8	GetIntegerv	KEEP	4.1.4	stencil-buffer
STENCIL_PASS_DEPTH_FAIL	Z8	GetIntegerv	KEEP	4.1.4	stencil-buffer
STENCIL_PASS_DEPTH_PASS	Z8	GetIntegerv	KEEP	4.1.4	stencil-buffer

NOTE: the only change is that Z6 type changes to Z8

New Implementation Dependent State

None

Name

EXT_texture3D

Name Strings

GL_EXT_texture3D

Version

$Date: 1996/04/05 19:17:05 $ $Revision: 1.22 $

Number

6

Dependencies

EXT_abgr affects the definition of this extension
EXT_texture is required

Overview

This extension defines 3-dimensional texture mapping. In order to define a 3D texture image conveniently, this extension also defines the in-memory formats for 3D images, and adds pixel storage modes to support them.

One important application of 3D textures is rendering volumes of image data.

New Procedures and Functions

```
void TexImage3DEXT(enum target,
                   int level,
                   enum internalformat,
                   sizei width,
                   sizei height,
                   sizei depth,
                   int border,
                   enum format,
                   enum type,
                   const void* pixels);
```

New Tokens

Accepted by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev, and by the <pname> parameter of PixelStore:

```
    PACK_SKIP_IMAGES_EXT             0x806B
    PACK_IMAGE_HEIGHT_EXT            0x806C
    UNPACK_SKIP_IMAGES_EXT           0x806D
    UNPACK_IMAGE_HEIGHT_EXT          0x806E
```

    Accepted by the <cap> parameter of Enable, Disable, and IsEnabled, by
    the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and
    GetDoublev, and by the <target> parameter of TexImage3DEXT, GetTexImage,
    GetTexLevelParameteriv, GetTexLevelParameterfv, GetTexParameteriv, and
    GetTexParameterfv:

```
        TEXTURE_3D_EXT                  0x806F
```

    Accepted by the <target> parameter of TexImage3DEXT,
    GetTexLevelParameteriv, and GetTexLevelParameterfv:

```
        PROXY_TEXTURE_3D_EXT            0x8070
```

    Accepted by the <pname> parameter of GetTexLevelParameteriv and
    GetTexLevelParameterfv:

```
        TEXTURE_DEPTH_EXT               0x8071
```

    Accepted by the <pname> parameter of TexParameteriv, TexParameterfv,
    GetTexParameteriv, and GetTexParameterfv:

```
        TEXTURE_WRAP_R_EXT              0x8072
```

    Accepted by the <pname> parameter of GetBooleanv, GetIntegerv,
    GetFloatv, and GetDoublev:

```
        MAX_3D_TEXTURE_SIZE_EXT         0x8073
```

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

None

Additions to Chapter 3 of the GL Specification (Rasterization)

The pixel storage modes are augmented to support 3D image formats in memory. Table 3.1 is replaced with the table below:

Parameter Name	Type	Initial Value	Valid Range
UNPACK_SWAP_BYTES	boolean	FALSE	TRUE/FALSE
UNPACK_LSB_FIRST	boolean	FALSE	TRUE/FALSE
UNPACK_ROW_LENGTH	integer	0	[0, infinity]
UNPACK_SKIP_ROWS	integer	0	[0, infinity]
UNPACK_SKIP_PIXELS	integer	0	[0, infinity]
UNPACK_ALIGNMENT	integer	4	1, 2, 4, 8
UNPACK_IMAGE_HEIGHT_EXT	integer	0	[0, infinity]
UNPACK_SKIP_IMAGES_EXT	integer	0	[0, infinity]

Table 3.1: PixelStore parameters pertaining to one or more of DrawPixels, TexImage1D, TexImage2D, and TexImage3DEXT.

When TexImage3DEXT is called, the groups in memory are treated as being arranged in a sequence of adjacent rectangles. Each rectangle is a 2-dimensional image, whose size and organization are specified by the <width> and <height> parameters to TexImage3DEXT. The values of UNPACK_ROW_LENGTH and UNPACK_ALIGNMENT control the row-to-row spacing in these images in exactly the manner described in the GL Specification for

2-dimensional images. If the value of UNPACK_IMAGE_HEIGHT_EXT is not positive, then the number of rows in each 2-dimensional image is <height>; otherwise the number of rows is UNPACK_IMAGE_HEIGHT_EXT. Each 2-dimensional image comprises an integral number of rows, and is exactly adjacent to its neighbor images.

The mechanism for selecting a sub-volume of a 3-dimensional image builds on the mechanism for selecting a sub-rectangle of groups from a larger containing rectangle. If UNPACK_SKIP_IMAGES_EXT is positive, the pointer is advanced by UNPACK_SKIP_IMAGES_EXT times the number of elements in one 2-dimensional image. Then <depth> 2-dimensional images are processed, each having a subimage extracted in the manner described in the GL Specification for 2-dimensional images.

The selected groups are processed as though they were part of a 2-dimensional image. When the final R, G, B, and A components have been computed for a group, they are assigned to components of a texel as described by Table 3.6 in the EXT_texture extension. Counting from zero, each resulting Nth texel is assigned internal integer coordinates [i,j,k], where

```
    i = (N mod width) - border

    j = ((N div width) mod height) - border

    k = ((N div (width * height)) mod depth) - border
```

and the div operator performs integer division with truncation. Thus the last 2-dimensional image of the 3-dimensional image is indexed with the highest value of k. The dimensions of the 3-dimensional texture image are <width> x <height> x <depth>. Integer values that will represent the base-2 logarithm of these dimensions are n, m, and l, defined such that

```
    width = 2**n + (2 * border)

    height = 2**m + (2 * border)

    depth = 2**l + (2 * border)
```

It is acceptable for an implementation to vary its allocation of internal component resolution based any TexImage3DEXT parameter, but the allocation must not be a function of any other factor, and cannot be changed once it is established. In particular, allocations must be invariant -- the same allocation must be made each time a texture image is specified with the same parameter values. Provision is made for an application to determine what component resolutions are available without having to fully specify the texture (see below).

Texture Wrap Modes

The additional token value TEXTURE_WRAP_R_EXT is accepted by TexParameteri, TexParameterv, TexParameteriv, and TexParameterfv, causing table 3.7 to be replaced with the table below:

Name	Type	Legal Values
TEXTURE_WRAP_S	integer	CLAMP, REPEAT
TEXTURE_WRAP_T	integer	CLAMP, REPEAT
TEXTURE_WRAP_R_EXT	integer	CLAMP, REPEAT
TEXTURE_MIN_FILTER	integer	NEAREST, LINEAR, NEAREST_MIPMAP_NEAREST, NEAREST_MIPMAP_LINEAR, LINEAR_MIPMAP_NEAREST, LINEAR_MIPMAP_LINEAR
TEXTURE_MAG_FILTER	integer	NEAREST, LINEAR
TEXTURE_BORDER_COLOR	4 floats	any 4 values in [0,1]

Table 3.7: Texture parameters and their values.

If TEXTURE_WRAP_R_EXT is set to REPEAT, then the GL ignores the integer part of R coordinates, using only the fractional part. CLAMP causes R to be clamped to the range [0, 1]. The initial state is for TEXTURE_WRAP_R_EXT to be REPEAT.

Texture Minification

Continuous coordinates s, t, u, and v are defined in figure 3.10 of the GL Specification. To discuss 3-dimensional texture mapping, coordinates r and w are defined similarly. Coordinate w is equal to -border at the "far" edge of the 3D image, understanding the image to be right-handed, with k values increasing toward the viewer. It has value depth+border at the near edge of this volume. Coordinate r has the same direction, but is normalized so that it is 0.0 and 1.0 at the "far" and "near" edges of a borderless volume. If the volume has a border, the 0.0 and 1.0 mappings of r continue to bound the core image.

The formulas for p, used to determine the level of detail, are modified by including dw/dx and dw/dy terms in the obvious ways. Equation 3.7 sums (dw/dx)**2 into the left term, and (dw/dy)**2 into the right term. Equation 3.8 has ((dw/dx * Dx + dw/dy * Dy)**2 added to the two terms under the square root. The requirements for the function f(x,y) become

1. f(x, y) is continuous and monotonically increasing in each of |du/dx|, |du/dy|, |dv/dx|, |dv/dy|, |dw/dx|, and |dw/dy|.

2. Let

```
m_u = max(|du/dx|, |du/dy|)
m_v = max(|dv/dx|, |dv/dy|)
m_w = max(|dw/dx|, |dw/dy|)
```

   Then

```
max(m_u, m_v, m_w) <= f(x, y) <= m_u + m_v + m_w
```

The i and j coordinates of the texel selected for NEAREST filtering are as defined in equations 3.9 and 3.10 of the GL Specification. Coordinate k is computed as

```
          /  floor(w),       r < 1
    k = (
          \  2**l - 1,       r = 1
```

A 2x2x2 cube of texels is selected for LINEAR filtering. The i and j coordinates of these texels are computed as defined in the GL Specification for 2-dimensional images. The k coordinates are computed as

```
           / floor(w - 1/2) mod 2**l,        TEXTURE_WRAP_R_EXT is REPEAT
    k0 = (
           \ floor(w - 1/2),                 TEXTURE_WRAP_R_EXT is CLAMP


           / (k0 + 1) mod 2**l,      TEXTURE_WRAP_R_EXT is REPEAT
    k1 = (
           \ k0 + 1,                 TEXTURE_WRAP_R_EXT is CLAMP
```

Let

```
    A = frac(u - 1/2)
    B = frac(v - 1/2)
    C = frac(w - 1/2)
```

where frac(x) denotes the fractional part of x. Let T[i,j,k] be the texel at location [i,j,k] in the texture image. Then the texture value, T, is found as

```
    T = (1-A) * (1-B) * (1-C) * T[i0,j0,k0] +
          A   * (1-B) * (1-C) * T[i1,j0,k0] +
        (1-A) *   B   * (1-C) * T[i0,j1,k0] +
          A   *   B   * (1-C) * T[i1,j1,k0] +
        (1-A) * (1-B) *   C   * T[i0,j0,k1] +
          A   * (1-B) *   C   * T[i1,j0,k1] +
        (1-A) *   B   *   C   * T[i0,j1,k1] +
          A   *   B   *   C   * T[i1,j1,k1]
```

for a 3-dimensional texture. If any of the selected T[i,j,k] in the above equation refer to a border texel with unspecified value, then the border color given by the current setting of TEXTURE_BORDER_COLOR is used instead of the unspecified value or values.

Mipmapping

TEXTURE_MIN_FILTER values NEAREST_MIPMAP_NEAREST, NEAREST_MIPMAP_LINEAR, LINEAR_MIPMAP_NEAREST, and LINEAR_MIPMAP_LINEAR each require the use of a mipmap. A 3-dimensional mipmap is an ordered set of arrays representing the same image; each array has a resolution lower than the previous one. If the texture, excluding is border, has dimensions 2**n x 2**m x 2**l, then there are exactly max(n, m, l) + 1 mipmap arrays. Each subsequent array has dimensions

```
    size(i-1) x size(j-1) x size(k-1)
```

where the dimensions of the previous array are

```
    size(i) x size(j) x size(k)
```

and

```
               /  2**x + 2*border,   x > 0
    size(x) = (
               \  1 + 2*border,      x <= 0
```

Each array in a 3-dimensional mipmap is transmitted to the GL using TexImage3DEXT; the array being set is indicated with the <level> parameter. The rules for completeness of the set of arrays are as described in the GL Specification, augmented in EXT_texture. The rules for mipmap array selection, and for filtering of the two selected arrays, are also as described in the GL Specification. Finally, the rules for texture magnification are also exactly as described in the GL Specification.

Texture Application

3-dimensional texture mapping is enabled and disabled using the generic Enable and Disable commands, with <cap> specified as TEXTURE_3D_EXT. If either or both TEXTURE_1D or TEXTURE_2D are enabled at the same time as TEXTURE_3D_EXT, the 3-dimensional texture is used.

Query support

The proxy texture PROXY_TEXTURE_3D_EXT can be used by applications to query an implementations maximum configurations just as it can be for 1-dimensional and 2-dimensional textures.

Alternate sets of partial per-level texture state are defined for the proxy texture PROXY_TEXTURE_3D_EXT. Specifically, TEXTURE_WIDTH, TEXTURE_HEIGHT, TEXTURE_DEPTH_EXT, TEXTURE_BORDER, TEXTURE_COMPONENTS, TEXTURE_RED_SIZE_EXT, TEXTURE_GREEN_SIZE_EXT, TEXTURE_BLUE_SIZE_EXT, TEXTURE_ALPHA_SIZE_EXT, TEXTURE_LUMINANCE_SIZE_EXT, and TEXTURE_INTENSITY_SIZE_EXT are maintained the the proxy texture. When TexImage3DEXT is called with <target> set to PROXY_TEXTURE_3D_EXT, these proxy state values are always respecified, even if the texture is too large to actually be used. If the texture is too large, all of these state variables are set to zero. If the texture could be accommodated by TexImage3DEXT called with <target> TEXTURE_3D_EXT, these values are set as though TEXTURE_3D_EXT were being defined. All of these state value can be queried with GetTexLevelParameteriv with <target> set to PROXY_TEXTURE_3D_EXT. Calling TexImage3DEXT with <target> PROXY_TEXTURE_3D_EXT has no effect on the actual 3-dimensional texture or its state.

There is no image associated with PROXY_TEXTURE_3D_EXT. Therefore PROXY_TEXTURE_3D_EXT cannot be used as a texture, and its image must never be queried using GetTexImage. (The error INVALID_ENUM results if this is attempted.) Likewise, there is no nonlevel-related state associated with a proxy texture, so calling GetTexParameteriv or GetTexParameterfv with <target> PROXY_TEXTURE_3D_EXT results in the error INVALID_ENUM.

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the GL Specification (Special Functions)

TexImage3DEXT with a proxy target is not included in display lists, but is instead executed immediately.

Additions to Chapter 6 of the GL Specification (State and State Requests)

3-dimensional texture images are queried using GetTexImage with its <target> parameter set to TEXTURE_3D_EXT. The assignment of texel component values to the initial R, G, B, and A components of a pixel group is described in EXT_texture. Pixel transfer and pixel storage operations are applied as if the image were 2-dimensional, except that the additional pixel storage state values PACK_IMAGE_HEIGHT_EXT and PACK_SKIP_IMAGES_EXT affect the storage of the image into memory. The correspondence of texels to memory locations is as defined for TexImage3DEXT above, substituting PACK* state for UNPACK* state in all occurrences.

Additions to the GLX Specification

None

GLX Protocol

A new GL rendering command is added. This command contains pixel data; thus it is sent to the server either as part of a glXRender request or as part of a glXRenderLarge request:

```
   TexImage3DEXT
       2           84+n+p          rendering command length
       2           4114            rendering command opcode
       1           BOOL            swap_bytes
       1           BOOL            lsb_first
       2                           unused
       4           CARD32          row_length
       4           CARD32          image_height
       4           CARD32          image_depth
       4           CARD32          skip_rows
       4           CARD32          skip_images
       4           CARD32          skip_volumes
       4           CARD32          skip_pixels
       4           CARD32          alignment
       4           ENUM            target
       4           INT32           level
       4           ENUM            internalformat
       4           INT32           width
       4           INT32           height
       4           INT32           depth
       4           INT32           size4d
       4           INT32           border
       4           ENUM            format
       4           ENUM            type
       4           CARD32          null_image
       n           LISTofBYTE      pixels
       p                           unused, p=pad(n)

       If the command is encoded in a glXRenderLarge request, the command
       opcode and command length fields above are expanded to 4 bytes each:

       4           88+n+p          rendering command length
       4           4114            rendering command opcode
```

If <width> < 0, <height> < 0, <depth> < 0, <format> is invalid or <type> is invalid, then the command is erroneous and n=0.

<pixels> is arranged as a sequence of adjacent rectangles. Each rectangle is a 2-dimensional image, whose structure is determined by the image height and the parameters <swap_bytes>, <lsb_first>, <row_length>, <skip_rows>, <skip_pixels>, <alignment>, <width>, <format>, and <type> given in the request. If <image_height> is not positive then the number of rows (i.e., the image height) is <height>; otherwise the number of rows is <image_height>.

<skip_images> allows a sub-volume of the 3-dimensional image to be selected. If <skip_images> is positive, then the pointer is advanced by <skip_images> times the number of elements in one 2-dimensional image. Then <depth> 2-dimensional images are read, each having a subimage extracted in the manner described in Appendix A of the GLX Protocol Specification.

Dependencies on EXT_abgr

If EXT_abgr is supported, the <format> parameter of TexImage3DEXT accepts ABGR_EXT. Otherwise it does not.

Dependencies on EXT_texture

EXT_texture is required. All of the <components> tokens defined by EXT_texture are accepted by the <internalformat> parameter of TexImage3DEXT, with the same semantics that are defined by EXT_texture.

The query and error extensions defined by EXT_texture are extended in this document.

Errors

INVALID_ENUM is generated if <target> is not TEXTURE_3D_EXT or PROXY_TEXTURE_3D_EXT.

INVALID_ENUM is generated if the <target> parameter to GetTexParameteriv, GetTexParameterfv or GetTexImage is PROXY_TEXTURE_3D_EXT.

INVALID_VALUE is generated if <level> is less than zero

INVALID_ENUM is generated if <internalformat> is not ALPHA, RGB, RGBA, LUMINANCE, LUMINANCE_ALPHA, or one of the tokens defined by the EXT_texture extension. (Values 1, 2, 3, and 4 are not accepted as internal formats by TexImage3DEXT).

INVALID_VALUE is generated if <width>, <height>, or <depth> is less than zero, or cannot be represented as 2**k + 2*border for some integer k.

INVALID_VALUE is generated if <border> is not 0 or 1.

INVALID_ENUM is generated if <format> is not COLOR_INDEX, RED, GREEN, BLUE, ALPHA, RGB, RGBA, LUMINANCE, or LUMINANCE_ALPHA (or ABGR_EXT if EXT_abgr is supported).

INVALID_ENUM is generated if <type> is not UNSIGNED_BYTE, BYTE, UNSIGNED_SHORT, SHORT, UNSIGNED_INT, INT, or FLOAT.

INVALID_OPERATION is generated if TexImage3DEXT is called between execution of Begin and the corresponding execution of End.

TEXTURE_TOO_LARGE_EXT is generated if the texture as specified cannot be accommodated by the implementation. This error will not occur if none of <width>, <height>, or <depth> is greater than MAX_3D_TEXTURE_SIZE_EXT.

New State

Get Value	Get Command	Type	Initial Value	Attribute
UNPACK_SKIP_IMAGES_EXT	GetIntegerv	Z+	0	-
UNPACK_IMAGE_HEIGHT_EXT	GetIntegerv	Z+	0	-
PACK_SKIP_IMAGES_EXT	GetIntegerv	Z+	0	-
PACK_IMAGE_HEIGHT_EXT	GetIntegerv	Z+	0	-
TEXTURE_3D_EXT	IsEnabled	B	FALSE	texture/enable
TEXTURE_WRAP_R_EXT	GetTexParameteriv	1 x Z2	REPEAT	texture
TEXTURE_DEPTH_EXT	GetTexLevelParameteriv	1 x 2 x levels x Z+	0	-

(old state with new type information)

Get Value	Get Command	Type	Initial Value	Attribute
TEXTURE	GetTexImage	3 x 1 x levels x I	null	-
TEXTURE_RED_SIZE_EXT	GetTexLevelParameteriv	3 x 2 x levels x Z+	0	-
TEXTURE_GREEN_SIZE_EXT	GetTexLevelParameteriv	3 x 2 x levels x Z+	0	-
TEXTURE_BLUE_SIZE_EXT	GetTexLevelParameteriv	3 x 2 x levels x Z+	0	-
TEXTURE_ALPHA_SIZE_EXT	GetTexLevelParameteriv	3 x 2 x levels x Z+	0	-
TEXTURE_LUMINANCE_SIZE_EXT	GetTexLevelParameteriv	3 x 2 x levels x Z+	0	-
TEXTURE_INTENSITY_SIZE_EXT	GetTexLevelParameteriv	3 x 2 x levels x Z+	0	-
TEXTURE_WIDTH	GetTexLevelParameteriv	3 x 2 x levels x Z+	0	-
TEXTURE_HEIGHT	GetTexLevelParameteriv	2 x 2 x levels x Z+	0	-
TEXTURE_BORDER	GetTexLevelParameteriv	3 x 2 x levels x Z+	0	-
TEXTURE_COMPONENTS (1D and 2D)	GetTexLevelParameteriv	2 x 2 x levels x Z42	1	-
TEXTURE_COMPONENTS (3D)	GetTexLevelParameteriv	1 x 2 x levels x Z38	LUMINANCE	-
TEXTURE_BORDER_COLOR	GetTexParameteriv	3 x C	0, 0, 0, 0	texture
TEXTURE_MIN_FILTER	GetTexParameteriv	3 x Z6	NEAREST_MIPMAP_LINEAR	texture
TEXTURE_MAG_FILTER	GetTexParameteriv	3 x Z2	LINEAR	texture
TEXTURE_WRAP_S	GetTexParameteriv	3 x Z2	REPEAT	texture
TEXTURE_WRAP_T	GetTexParameteriv	2 x Z2	REPEAT	texture

New Implementation Dependent State

Get Value	Get Command	Type	Minimum Value
MAX_3D_TEXTURE_SIZE_EXT	GetIntegerv	Z+	16

Name

    EXT_texture_array

Name Strings

    GL_EXT_texture_array

Contact

Pat Brown, NVIDIA Corporation (pbrown 'at' nvidia.com)

Status

    Shipping for GeForce 8 Series (November 2006, Release 95)

Version

```
    Last Modified Date:         02/04/2008
    Author revision:            6
```

Number

    329

Dependencies

    This extension is written against the OpenGL 2.0 specification and version
    1.10.59 of the OpenGL Shading Language specification.

    This extension is interacts with EXT_framebuffer_object.

    This extension interacts with NV_geometry_program4.

    This extension interacts with NV_gpu_program4 or the OpenGL Shading
    Language, which provide the mechanisms necessary to access array textures.

    This extension interacts with EXT_texture_compression_s3tc and
    NV_texture_compression_vtc.

Overview

    This extension introduces the notion of one- and two-dimensional array
    textures. An array texture is a collection of one- and two-dimensional
    images of identical size and format, arranged in layers. A
    one-dimensional array texture is specified using TexImage2D; a
    two-dimensional array texture is specified using TexImage3D. The height
    (1D array) or depth (2D array) specify the number of layers in the image.

    An array texture is accessed as a single unit in a programmable shader,
    using a single coordinate vector. A single layer is selected, and that
    layer is then accessed as though it were a one- or two-dimensional
    texture. The layer used is specified using the "t" or "r" texture
    coordinate for 1D and 2D array textures, respectively. The layer
    coordinate is provided as an unnormalized floating-point value in the
    range [0,<n>-1], where <n> is the number of layers in the array texture.
    Texture lookups do not filter between layers, though such filtering can be

achieved using programmable shaders. When mipmapping is used, each level of an array texture has the same number of layers as the base level; the number of layers is not reduced as the image size decreases.

Array textures can be rendered to by binding them to a framebuffer object (EXT_framebuffer_object). A single layer of an array texture can be bound using normal framebuffer object mechanisms, or an entire array texture can be bound and rendered to using the layered rendering mechanisms provided by NV_geometry_program4.

This extension does not provide for the use of array textures with fixed-function fragment processing. Such support could be added by providing an additional extension allowing applications to pass the new target enumerants (TEXTURE_1D_ARRAY_EXT and TEXTURE_2D_ARRAY_EXT) to Enable and Disable.

New Procedures and Functions

```
void FramebufferTextureLayerEXT(enum target, enum attachment,
                                uint texture, int level, int layer);
```

New Tokens

Accepted by the <target> parameter of TexParameteri, TexParameteriv, TexParameterf, TexParameterfv, and BindTexture:

```
TEXTURE_1D_ARRAY_EXT                            0x8C18
TEXTURE_2D_ARRAY_EXT                            0x8C1A
```

Accepted by the <target> parameter of TexImage3D, TexSubImage3D, CopyTexSubImage3D, CompressedTexImage3D, and CompressedTexSubImage3D:

```
TEXTURE_2D_ARRAY_EXT
PROXY_TEXTURE_2D_ARRAY_EXT                      0x8C1B
```

Accepted by the <target> parameter of TexImage2D, TexSubImage2D, CopyTexImage2D, CopyTexSubImage2D, CompressedTexImage2D, and CompressedTexSubImage2D:

```
TEXTURE_1D_ARRAY_EXT
PROXY_TEXTURE_1D_ARRAY_EXT                      0x8C19
```

Accepted by the <pname> parameter of GetBooleanv, GetDoublev, GetIntegerv and GetFloatv:

```
TEXTURE_BINDING_1D_ARRAY_EXT                    0x8C1C
TEXTURE_BINDING_2D_ARRAY_EXT                    0x8C1D
MAX_ARRAY_TEXTURE_LAYERS_EXT                    0x88FF
```

```
    Accepted by the <param> parameter of TexParameterf, TexParameteri,
    TexParameterfv, and TexParameteriv when the <pname> parameter is
    TEXTURE_COMPARE_MODE_ARB:

        COMPARE_REF_DEPTH_TO_TEXTURE_EXT                0x884E

    (Note:  COMPARE_REF_DEPTH_TO_TEXTURE_EXT is simply an alias for the
    existing COMPARE_R_TO_TEXTURE token in OpenGL 2.0; the alternate name
    reflects the fact that the R coordinate is not always used.)

    Accepted by the <internalformat> parameter of TexImage3D and
    CompressedTexImage3D, and by the <format> parameter of
    CompressedTexSubImage3D:

        COMPRESSED_RGB_S3TC_DXT1_EXT
        COMPRESSED_RGBA_S3TC_DXT1_EXT
        COMPRESSED_RGBA_S3TC_DXT3_EXT
        COMPRESSED_RGBA_S3TC_DXT5_EXT

    Accepted by the <pname> parameter of
    GetFramebufferAttachmentParameterivEXT:

        FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER_EXT          0x8CD4

    (Note:  FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER is simply an alias for the
    FRAMEBUFFER_ATTACHMENT_TEXTURE_3D_ZOFFSET_EXT token provided in
    EXT_framebuffer_object.  This extension generalizes the notion of
    "<zoffset>" to include layers of an array texture.)

    Returned by the <type> parameter of GetActiveUniform:

        SAMPLER_1D_ARRAY_EXT                             0x8DC0
        SAMPLER_2D_ARRAY_EXT                             0x8DC1
        SAMPLER_1D_ARRAY_SHADOW_EXT                      0x8DC3
        SAMPLER_2D_ARRAY_SHADOW_EXT                      0x8DC4
```

Additions to Chapter 2 of the OpenGL 2.0 Specification (OpenGL Operation)

Modify section 2.15.3, "Shader Variables", page 75

```
    Add the following new return types to the description of GetActiveUniform
    on p. 81.

      SAMPLER_1D_ARRAY_EXT,
      SAMPLER_2D_ARRAY_EXT,
      SAMPLER_1D_ARRAY_SHADOW_EXT,
      SAMPLER_2D_ARRAY_SHADOW_EXT
```

Modify Section 2.15.4, Shader Execution (p. 84)

```
    (modify first paragraph, p. 86 -- two simple edits:

      (1) Change reference to the "r" coordinate to simply indicate that the
          reference value for shadow mapping is provided in the lookup
          function.  It's still usually in the "r" coordinate, except for
          two-dimensional array textures, where it's in "q".
```

(2) Add new EXT_gpu_shader4 sampler types used for array textures.)

Texture lookups involving textures with depth component data can either return the depth data directly or return the results of a comparison with a reference depth value specified in the coordinates passed to the texture lookup function, as described in section 3.8.14. The comparison operation is requested in the shader by using the shadow sampler types (sampler1DShadow, sampler2DShadow, sampler1DArrayShadow, or sampler2DArrayShadow) and in the texture using the TEXTURE_COMPARE_MODE parameter. ...

Additions to Chapter 3 of the OpenGL 2.0 Specification (Rasterization)

Modify Section 3.8, Texturing (p. 149).

(add new paragraph at the top of p. 150) Six types of texture are supported; each is a collection of images built from one-, two-, or three-dimensional array of image elements referred to as texels. One-, two-, and three-dimensional textures consist of a one-, two-, or three-dimensional texel arrays. One- and two-dimensional array textures are arrays of one- or two-dimensional images, consisting of one or more layers. Finally, a cube map is a special two-dimensional array texture with six layers that represent the faces of a cube. When accessing a cube map, the texture coordinates are projected onto one of the six faces.

Modify Section 3.8.1, Texture Image Specification (p. 150).

(modify first paragraph of section, p. 150) The command

```
  void TexImage3D( enum target, int level, int internalformat,
                   sizei width, sizei height, sizei depth, int border,
                   enum format, enum type, void *data );
```

is used to specify a three-dimensional texture image. target must be one of TEXTURE_3D for a three-dimensional texture or TEXTURE_2D_ARRAY_EXT for an two-dimensional array texture. Additionally, target may be either PROXY_TEXTURE_3D for a three-dimensional proxy texture, or PROXY_TEXTURE_2D_ARRAY_EXT for a two-dimensional proxy array texture. ...

(modify the fourth paragraph on p. 151) Textures with a base internal format of DEPTH_COMPONENT are supported by texture image specification commands only if target is TEXTURE_1D, TEXTURE_2D, TEXTURE_1D_ARRAY_EXT, TEXTURE_2D_ARRAY_EXT, PROXY_TEXTURE_1D, PROXY_TEXTURE_2D, PROXY_TEXTURE_1D_ARRAY_EXT, or PROXY_TEXTURE_2D_ARRAY_EXT. Using this format in conjunction with any other target will result in an INVALID OPERATION error.

(modify the first paragraph on p. 153 -- In particular, add new terms w_b, h_b, and d_b to represent border width, height, or depth, instead of a single border size term b_s. Subsequent equations referring to b_s should be modified to refer to w_b, h_b, and d_b, as appropriate.)

```
... Counting from zero, each resulting Nth texel is assigned internal
integer coordinates (i, j, k), where

  i = (N mod width) - w_b
  j = (floor(N/width) mod height) - h_b
  k = (floor(N/(width*height)) mod depth) - d_b

and w_b, h_b, and d_b are the specified border width, height, and depth.
w_b and h_b are the specified <border> value; d_b is the specified
<border> value if <target> is TEXTURE_3D or zero if <target> is
TEXTURE_2D_ARRAY_EXT. ...

(modify equations 3.15-3.17 and third paragraph of p. 155)

  w_s = w_t + 2 * w_b                     (3.15)
  h_s = h_t + 2 * h_b                     (3.16)
  d_s = d_t + 2 * d_b                     (3.17)

... If <border> is less than zero, or greater than b_t, then the error
INVALID_VALUE is generated.

(modify the last paragraph on p. 155 on to p. 156)

The maximum allowable width, height, or depth of a texel array for a
three-dimensional texture is an implementation dependent function of the
level-of-detail and internal format of the resulting image array.  It must
be at least 2^(k-lod) + 2 * b_t for image arrays of level-of-detail 0
through k, where k is the log base 2 of MAX_3D_TEXTURE_SIZE, lod is the
level-of-detail of the image array, and b_t is the maximum border width.
It may be zero for image arrays of any level-of-detail greater than k. The
error INVALID VALUE is generated if the specified image is too large to be
stored under any conditions.

In a similar fashion, the maximum allowable width of a texel array for a
one- or two-dimensional, or one- or two-dimensional array texture, and the
maximum allowable height of a two-dimensional or two-dimensional array
texture, must be at least 2^(k-lod) + 2 * b_t for image arrays of level 0
through k, where k is the log base 2 of MAX_TEXTURE_SIZE. The maximum
allowable width and height of a cube map texture must be the same, and
must be at least 2^(k-lod) + 2 * b_t for image arrays level 0 through k,
where k is the log base 2 of MAX_CUBE_MAP_TEXTURE_SIZE.  The maximum
number of layers for one- and two-dimensional array textures (height or
depth, respectively) must be at least MAX_ARRAY_TEXTURE_LAYERS_EXT for all
levels.

(modify the fourth paragraph on p. 156) The command

  void TexImage2D( enum target, int level,
                   int internalformat, sizei width, sizei height,
                   int border, enum format, enum type, void *data );

is used to specify a two-dimensional texture image. target must be one of
TEXTURE_2D for a two-dimensional texture, TEXTURE_1D_ARRAY_EXT for a
one-dimensional array texture, or one of TEXTURE_CUBE_MAP_POSITIVE_X,
TEXTURE_CUBE_MAP_NEGATIVE_X, TEXTURE_CUBE_MAP_POSITIVE_Y,
TEXTURE_CUBE_MAP_NEGATIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z, or
TEXTURE_CUBE_MAP_NEGATIVE_Z for a cube map texture. Additionally, target
```

may be either PROXY_TEXTURE_2D for a two-dimensional proxy texture, PROXY_TEXTURE_1D_ARRAY_EXT for a one-dimensional proxy array texture, or PROXY TEXTURE_CUBE_MAP for a cube map proxy texture in the special case discussed in section 3.8.11. The other parameters match the corresponding parameters of TexImage3D.

For the purposes of decoding the texture image, TexImage2D is equivalent to calling TexImage3D with corresponding arguments and depth of 1, except that

* The border depth, d_b, is zero, and the depth of the image is always 1 regardless of the value of border.

* The border height, h_b, is zero if <target> is TEXTURE_1D_ARRAY_EXT, and <border> otherwise.

* Convolution will be performed on the image (possibly changing its width and height) if SEPARABLE 2D or CONVOLUTION 2D is enabled.

* UNPACK SKIP IMAGES is ignored.

(modify the fourth paragraph on p. 157) For the purposes of decoding the texture image, TexImage1D is equivalent to calling TexImage2D with corresponding arguments and height of 1, except that

* The border height and depth (h_b and d_b) are always zero, regardless of the value of <border>.

* Convolution will be performed on the image (possibly changing its width) only if CONVOLUTION 1D is enabled.

(modify the last paragraph on p. 157 and the first paragraph of p. 158 -- changing the phrase "texture array" to "texel array" to avoid confusion with array textures. All subsequent references to "texture array" in the specification should also be changed to "texel array".)

We shall refer to the (possibly border augmented) decoded image as the texel array. A three-dimensional texel array has width, height, and depth ws, hs, and ds as defined respectively in equations 3.15, 3.16, and 3.17. A two-dimensional texel array has depth ds = 1, with height hs and width ws as above, and a one-dimensional texel array has depth ds = 1, height hs = 1, and width ws as above.

An element (i,j,k) of the texel array is called a texel (for a two-dimensional texture or one-dimensional array texture, k is irrelevant; for a one-dimensional texture, j and k are both irrelevant). The texture value used in texturing a fragment is determined by that fragment’s associated (s,t,r) coordinates, but may not correspond to any actual texel. See figure 3.10.

Modify Section 3.8.2, Alternate Texture Image Specification Commands (p. 159)

(modify second paragraph, p. 159 -- allow 1D array textures) The command

```
  void CopyTexImage2D( enum target, int level,
                       enum internalformat, int x, int y, sizei width,
                       sizei height, int border );
```

defines a two-dimensional texture image in exactly the manner of TexImage2D, except that the image data are taken from the framebuffer rather than from client memory. Currently, target must be one of TEXTURE_2D, TEXTURE_1D_ARRAY_EXT, TEXTURE_CUBE_MAP_POSITIVE_X, TEXTURE_CUBE_MAP_NEGATIVE_X, TEXTURE_CUBE MAP_POSITIVE_Y, TEXTURE_CUBE_MAP_NEGATIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z, or TEXTURE_CUBE_MAP_NEGATIVE_Z.

(modify last paragraph, p. 160) ... Currently the target arguments of TexSubImage1D and CopyTexSubImage1D must be TEXTURE_1D, the target arguments of TexSubImage2D and CopyTexSubImage2D must be one of TEXTURE_2D, TEXTURE_1D_ARRAY_EXT, TEXTURE_CUBE_MAP_POSITIVE_X, TEXTURE_CUBE_MAP_NEGATIVE_X, TEXTURE_CUBE_MAP_POSITIVE_Y, TEXTURE_CUBE_MAP_NEGATIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z, or TEXTURE_CUBE_MAP_NEGATIVE_Z, and the target arguments of TexSubImage3D and CopyTexSubImage3D must be TEXTURE_3D or TEXTURE_2D_ARRAY_EXT. ...

(modify last paragraph, p. 161 and subsequent inequalities)

Negative values of xoffset, yoffset, and zoffset correspond to the coordinates of border texels, addressed as in figure 3.10. Taking w_s, h_s, d_s, w_b, h_b, and d_b to be the specified width, height, depth, and border width, height, and depth of the texture array, and taking x, y, z, w, h, and d to be the xoffset, yoffset, zoffset, width, height, and depth argument values, any of the following relationships generates the error INVALID VALUE:

```
    x < -w_b
    x + w > w_s - w_b
    y < -h_b
    y + h > h_s - h_b
    z < -d_b
    z + d > d_s - d_b
```

Modify Section 3.8.4, Texture Parameters (p. 166)

(modify first paragraph of section, p. 166) Various parameters control how the texel array is treated when specified or changed, and when applied to a fragment. Each parameter is set by calling

```
  void TexParameter{if}( enum target, enum pname, T param );
  void TexParameter{if}v( enum target, enum pname, T params );
```

target is the target, either TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP, TEXTURE_1D_ARRAY_EXT, or TEXTURE_2D_ARRAY_EXT.

Modify Section 3.8.8, Texture Minification (p. 170)

(modify first paragraph, p. 172) ... For a one-dimensional or one-dimensional array texture, define v(x, y) == 0 and w(x, y) == 0; for a two-dimensional, two-dimensional array, or cube map texture, define w(x, y) == 0. ...

(modify second paragraph, p. 173) For one-dimensional or one-dimensional array textures, j and k are irrelevant; the texel at location i becomes the texture value. For two-dimensional, two-dimensional array, or cube map textures, k is irrelevant; the texel at location (i, j) becomes the texture value. For one- and two-dimensional array textures, the texel is obtained from image layer l, where

```
  l = clamp(floor(t + 0.5), 0, h_t-1), for one-dimensional array textures,
      clamp(floor(r + 0.5), 0, d_t-1), for two-dimensional array textures.
```

(modify third paragraph, p. 174) For a two-dimensional, two-dimensional array, or cube map texture,

```
  tau = ...
```

where tau_ij is the texel at location (i, j) in the two-dimensional texture image. For two-dimensional array textures, all texels are obtained from layer l, where

```
  l = clamp(floor(r + 0.5), 0, d_t-1).
```

And for a one-dimensional or one-dimensional array texture,

```
  tau = ...
```

where tau_i is the texel at location i in the one-dimensional texture. For one-dimensional array textures, both texels are obtained from layer l, where

```
  l = clamp(floor(t + 0.5), 0, h_t-1).
```

(modify first two paragraphs of "Mipmapping", p. 175) TEXTURE_MIN_FILTER values NEAREST_MIPMAP_NEAREST, NEAREST_MIPMAP_LINEAR, LINEAR_MIPMAP_NEAREST, and LINEAR_MIPMAP_LINEAR each require the use of a mipmap. A mipmap is an ordered set of arrays representing the same image; each array has a resolution lower than the previous one.

If the image array of level level_base, excluding its border, has dimensions, w_t × h_t × d_t, then there are floor(log2(maxsize)) + 1 levels in the mipmap, where

```
  maxsize = w_t,                  for one-dimensional and one-dimensional
                                  array textures,
            max(w_t, h_t),        for two-dimensional, two-dimensional
                                  array, and cube map textures
            max(w_t, h_t, d_t),   for three dimensional textures.
```

Numbering the levels such that level level_base is the 0th level, the ith array has dimensions

```
  max(1, floor(w_t/w_d)) x max(1, floor(h_t/h_d)) x max(1, floor(d_t/d_d))
```

where

```
  w_d = 2 ^ i;
  h_d = 1, for one-dimensional array textures and
        2 ^ i, otherwise; and
  d_d = 1, for two-dimensional array textures and
        2 ^ i, otherwise,
```

until the last array is reached with dimension 1 × 1 × 1.

Each array in a mipmap is defined using TexImage3D, TexImage2D, CopyTexImage2D, TexImage1D, or CopyTexImage1D; the array being set is indicated with the level-of-detail argument level. Level-of-detail numbers proceed from level_base for the original texture array through p = floor(log2(maxsize)) + level_base with each unit increase indicating an array of half the dimensions of the previous one (rounded down to the next integer if fractional) as already described. All arrays from level_base through q = min{p, level_max} must be defined, as discussed in section 3.8.10.

(modify third paragraph in the "Mipmap Generation" section, p. 176)

The contents of the derived arrays are computed by repeated, filtered reduction of the level_base array. For one- and two-dimensional array textures, each layer is filtered independently. ...

Modify Section 3.8.10, Texture Completeness (p. 177)

(modify second paragaph of section, p. 177) For one-, two-, or three-dimensional textures and one- or two-dimensional array textures, a texture is complete if the following conditions all hold true: ...

Modify Section 3.8.11, Texture State and Proxy State (p. 178)

(modify second and third paragraphs, p. 179, adding array textures and making minor wording changes)

In addition to image arrays for one-, two-, and three-dimensional textures, one- and two-dimensional array textures, and the six image arrays for the cube map texture, partially instantiated image arrays are maintained for one-, two-, and three-dimensional textures and one- and two-dimensional array textures. Additionally, a single proxy image array is maintained for the cube map texture. Each proxy image array includes width, height, depth, border width, and internal format state values, as well as state for the red, green, blue, alpha, luminance, and intensity component resolutions. Proxy image arrays do not include image data, nor do they include texture properties. When TexImage3D is executed with target specified as PROXY_TEXTURE_3D, the three-dimensional proxy state values of the specified level-of-detail are recomputed and updated. If the image array would not be supported by TexImage3D called with target set to TEXTURE 3D, no error is generated, but the proxy width, height, depth, border width, and component resolutions are set to zero. If the image array would be supported by such a call to TexImage3D, the proxy state values are set exactly as though the actual image array were being specified. No pixel data are transferred or processed in either case.

Proxy arrays for one- and two-dimensional textures and one- and two-dimensional array textures are operated on in the same way when TexImage1D is executed with target specified as PROXY_TEXTURE_1D, TexImage2D is executed with target specified as PROXY_TEXTURE_2D or PROXY_TEXTURE_1D_ARRAY_EXT, or TexImage3D is executed with target specified as PROXY_TETXURE_2D_ARRAY_EXT.

Modify Section 3.8.12, Texture Objects (p. 180)

(update most of the beginning of the section to allow array textures)

In addition to the default textures TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP, TEXTURE_1D_ARRAY_EXT, and TEXTURE_2D_EXT, named one-, two-, and three-dimensional, cube map, and one- and two-dimensional array texture objects can be created and operated upon. The name space for texture objects is the unsigned integers, with zero reserved by the GL.

A texture object is created by binding an unused name to TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP, TEXTURE_1D_ARRAY_EXT, or TEXTURE_2D_ARRAY_EXT. The binding is effected by calling

```
  void BindTexture( enum target, uint texture );
```

with <target> set to the desired texture target and <texture> set to the unused name. The resulting texture object is a new state vector, comprising all the state values listed in section 3.8.11, set to the same initial values. If the new texture object is bound to TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP, TEXTURE_1D_ARRAY_EXT, or TEXTURE_2D_ARRAY_EXT, it is and remains a one-, two-, three-dimensional, cube map, one- or two-dimensional array texture respectively until it is deleted.

BindTexture may also be used to bind an existing texture object to either TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP, TEXTURE_1D_ARRAY_EXT, or TEXTURE_2D_ARRAY_EXT. The error INVALID_OPERATION is generated if an attempt is made to bind a texture object of different dimensionality than the specified target. If the bind is successful no change is made to the state of the bound texture object, and any previous binding to target is broken.

While a texture object is bound, GL operations on the target to which it is bound affect the bound object, and queries of the target to which it is bound return state from the bound object. If texture mapping of the dimensionality of the target to which a texture object is bound is enabled, the state of the bound texture object directs the texturing operation.

In the initial state, TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP, TEXTURE_1D_ARRAY_EXT, and TEXTURE_2D_ARRAY_EXT have one-, two-, three-dimensional, cube map, and one- and two-dimensional array texture state vectors respectively associated with them. In order that access to these initial textures not be lost, they are treated as texture objects all of whose names are 0. The initial one-, two-, three-dimensional, cube map, one- and two-dimensional array textures are therefore operated upon, queried, and applied as TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP, TEXTURE_1D_ARRAY_EXT, and TEXTURE_2D_ARRAY_EXT respectively while 0 is bound to the corresponding targets.

(modify second paragraph, p. 181) ... If a texture that is currently bound to one of the targets TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP, TEXTURE_1D_ARRAY_EXT, or TEXTURE_2D_ARRAY_EXT is deleted, it is as though BindTexture had been executed with the same target and texture zero. ...

(modify second paragraph, p. 182) The texture object name space, including the initial one-, two-, and three dimensional, cube map, and one- and two-dimensional array texture objects, is shared among all texture units. ...

Modify Section 3.8.14, Texture Comparison Modes (p. 185)

(modify second through fourth paragraphs, p. 188, reflecting that the texture coordinate used for depth comparisons varies, including a new enum name)

Let D_t be the depth texture value, in the range [0, 1]. For fixed-function texture lookups, let R be the interpolated <r> texture coordinate, clamped to the range [0, 1]. For texture lookups generated by a program instruction, let R be the reference value for depth comparisons provided in the instruction, also clamped to [0, 1]. Then the effective texture value L_t, I_t, or A_t is computed as follows: ...

If the value of TEXTURE_COMPARE_MODE is NONE, then

```
  r = Dt
```

If the value of TEXTURE_COMPARE_MODE is COMPARE_REF_DEPTH_TO_TEXTURE_EXT), then r depends on the texture comparison function as shown in table 3.27.

Modify Section 3.11.2, Shader Execution (p. 194)

(modify second paragraph, p. 195 -- two simple edits:

(1) Change reference to the "r" coordinate to simply indicate that the reference value for shadow mapping is provided in the lookup function. It's still usually in the "r" coordinate, except for two-dimensional array textures, where it's in "q".
(2) Add new EXT_gpu_shader4 sampler types used for array textures.)

Texture lookups involving textures with depth component data can either return the depth data directly or return the results of a comparison with a reference depth value specified in the coordinates passed to the texture lookup function. The comparison operation is requested in the shader by using the shadow sampler types (sampler1DShadow, sampler2DShadow, sampler1DArrayShadow, and sampler2DArrayShadow) and in the texture using the TEXTURE COMPARE MODE parameter. ...

Additions to Chapter 4 of the OpenGL 2.0 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 2.0 Specification (Special Functions)

Modify Section 5.4, Display Lists (p. 237)

(modify first paragraph, p. 242) TexImage3D, TexImage2D, TexImage1D, Histogram, and ColorTable are executed immediately when called with the corresponding proxy arguments PROXY_TEXTURE_3D or PROXY_TEXTURE_2D_ARRAY_EXT; PROXY_TEXTURE_2D, PROXY_TEXTURE_CUBE_MAP, or PROXY_TEXTURE_1D_ARRAY_EXT; PROXY_TEXTURE_1D; PROXY_HISTOGRAM; and PROXY_COLOR_TABLE, PROXY_POST_CONVOLUTION_COLOR_TABLE, or PROXY_POST_COLOR_MATRIX_COLOR_TABLE.

Additions to Chapter 6 of the OpenGL 2.0 Specification (State and State Requests)

Modify Section 6.1.3, Enumerated Queries (p. 246)

(modify second paragraph, p. 247)

GetTexParameter parameter <target> may be one of TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP, TEXTURE_1D_ARRAY_EXT, or TEXTURE_2D_ARRAY_EXT, indicating the currently bound one-, two-, three-dimensional, cube map, or one- or two-dimensional array texture. GetTexLevelParameter parameter target may be one of TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP_POSITIVE_X, TEXTURE_CUBE_MAP_NEGATIVE_X, TEXTURE_CUBE_MAP_POSITIVE_Y, TEXTURE_CUBE_MAP_NEGATIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z, TEXTURE_CUBE_MAP_NEGATIVE_Z, TEXTURE_1D_ARRAY_EXT, TEXTURE_2D_ARRAY_EXT, PROXY_TEXTURE_1D, PROXY_TEXTURE_2D, PROXY_TEXTURE_3D, PROXY_TEXTURE_CUBE_MAP, PROXY_TEXTURE_1D_ARRAY, or PROXY_TEXTURE_2D_ARRAY, indicating the one-, two-, or three-dimensional texture, one of the six distinct 2D images making up the cube map texture, the one- or two-dimensional array texture, or the one-, two-, three-dimensional, cube map, or one- or two-dimensional array proxy state vector. ...

Modify Section 6.1.4, Texture Queries (p. 248)

(modify first three paragraphs of section, p. 248) The command

```
  void GetTexImage( enum tex, int lod, enum format,
                    enum type, void *img );
```

is used to obtain texture images. It is somewhat different from the other get commands; tex is a symbolic value indicating which texture (or texture face in the case of a cube map texture target name) is to be obtained. TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_1D_ARRAY_EXT, and TEXTURE_2D_ARRAY_EXT indicate a one-, two-, or three-dimensional texture, or one- or two-dimensional array texture, respectively. TEXTURE_CUBE_MAP_POSITIVE_X, ...

GetTexImage obtains... from the first image to the last for three-dimensional textures. One- and two-dimensional array textures are treated as two- and three-dimensional images, respectively, where the layers are treated as rows or images. These groups are then...

For three-dimensional and two-dimensional array textures, pixel storage operations are applied as if the image were two-dimensional, except that the additional pixel storage state values PACK_IMAGE_HEIGHT and PACK_SKIP_IMAGES are applied. ...

Additions to Appendix A of the OpenGL 2.0 Specification (Invariance)

None.

Additions to the AGL/GLX/WGL Specifications

None.

GLX Protocol

None.

Dependencies on EXT_framebuffer_object

If EXT_framebuffer_object is supported, a single layer of an array texture can be bound to a framebuffer attachment point, and manual mipmap generation support is extended to include array textures.

Several modifications are made to the EXT_framebuffer_object specification. First, the token identifying the attached layer of a 3D texture, FRAMEBUFFER_ATTACHMENT_TEXTURE_3D_ZOFFSET_EXT, is renamed to FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER_EXT. This is done because this extension generalizes the "z offset" concept to become notion of attaching a layer of a multi-layer texture, which is applicable for both three-dimensional and array textures. All references to this token in EXT_framebuffer_object should be changed to the new token, and references to "z offset" in the specification text should be replaced with "layer" as appropriate. Additional edits follow.

(modify "Manual Mipmap Generation" in edits to Section 3.8.8)

Mipmaps can be generated manually with the command

```
  void GenerateMipmapEXT(enum target);
```

where <target> is one of TEXTURE_1D, TEXTURE_2D, TEXTURE_CUBE_MAP, TEXTURE_3D, TEXTURE_1D_ARRAY, or TEXTURE_2D_ARRAY. Mipmap generation affects the texture image attached to <target>. ...

(modify Section 4.4.2.3, Attaching Texture Images to a Framebuffer -- add to the end of the section)

The command

```
  void FramebufferTextureLayerEXT(enum target, enum attachment,
                                  uint texture, int level, int layer);
```

operates identically to FramebufferTexture3DEXT, except that it attaches a single layer of a three-dimensional texture or a one- or two-dimensional array texture. <layer> is an integer indicating the layer number, and is treated identically to the <zoffset> parameter in FramebufferTexture3DEXT. The error INVALID_VALUE is generated if <layer> is negative. The error INVALID_OPERATION is generated if <texture> is non-zero and is not the name of a three dimensional texture or one- or two-dimensional array texture. Unlike FramebufferTexture3D, no <textarget> parameter is accepted.

If <texture> is non-zero and the command does not result in an error, the framebuffer attachment state corresponding to <attachment> is updated as in the other FramebufferTexture commands, except that FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER_EXT is set to <layer>.

(modify Section 4.4.4.1, Framebuffer Attachment Completeness)

The framebuffer attachment point <attachment> is said to be "framebuffer attachment complete" if ...:

...

* If FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE_EXT is TEXTURE and FRAMEBUFFER_ATTACHMENT_OBJECT_NAME_EXT names a one- or two-dimensional array texture, then FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER_EXT must be smaller than the number of layers in the texture.

(modify Section 6.1.3, Enumerated Queries)

...

If <pname> is FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER_EXT and the texture object named FRAMEBUFFER_ATTACHMENT_OBJECT_NAME_EXT is a three-dimensional texture or a one- or two-dimensional array texture, then <params> will contain the number of texture layer attached to the attachment point. Otherwise, <params> will contain the value zero.

Dependencies on NV_geometry_program4

NV_geometry_program4 provides additional modifications to EXT_framebuffer_object to support layered rendering, which allows applications to bind entire three-dimensional, cube map, or array textures to a single attachment point, and select a layer to render to according to a layer number written by the geometry program.

The framebuffer object modifications provided in NV_geometry_program4 are more extensive than the more limited support provided for array textures. The edits in this spec are a functional subset of the edits in NV_geometry_program4. All of the modifications that this extension makes to EXT_framebuffer_object are superseded by NV_geometry_program4, except for the minor language changes made to GenerateMipmapsEXT().

Dependencies on NV_gpu_program4 and the OpenGL Shading Language (GLSL)

If NV_gpu_program4, EXT_gpu_shader4, and the OpenGL Shading Language (GLSL) are not supported, and no other mechanism is provided to perform texture lookups into array textures, this extension is pointless, given that it provides no fixed-function mechanism to access texture arrays.

If GLSL is supported, the language below describes the modifications to the shading language to support array textures. The extension EXT_gpu_shader4 provides a broader set of shading language modifications that include array texture lookup functions described here, plus a number of additional functions.

If GLSL is not supported, the shading language below and references to the SAMPLER_{1D,2D}_ARRAY_EXT and SAMPLER_{1D,2D}_ARRAY_SHADOW_EXT tokens should be removed.

Dependencies on EXT_texture_compression_s3tc and NV_texture_compression_vtc

S3TC texture compression is supported for two-dimensional array textures. When <target> is TEXTURE_2D_ARRAY_EXT, each layer is stored independently as a compressed two-dimensional textures. When specifying or querying compressed images using one of the S3TC formats, the images are provided and/or returned as a series of two-dimensional textures stored consecutively in memory, with the layer closest to zero specified first. For array textures, images are not arranged in 4x4x4 or 4x4x2 blocks as in the three-dimensional compression format provided in the EXT_texture_compression_vtc extension. Pixel store parameters, including those specific to three-dimensional images, are ignored when compressed image data are provided or returned, as in the EXT_texture_compression_s3tc extension.

S3TC compression is not supported for one-dimensional texture targets in EXT_texture_compression_s3tc, and is not supported for one-dimensional array textures in this extension. If compressed one-dimensional arrays are needed, use a two-dimensional texture with a height of one.

As with NV_texture_compression_vtc, this extension allows the use of the four S3TC internal format types in TexImage3D, CompressedTexImage3D, and CompressedTexSubImage3D calls. Unlike NV_texture_compression_vtc (for 3D textures), compressed sub-image updates are allowed at arbitrary locations along the Z axis. The language describing CompressedTexSubImage* APIs, edited by EXT_texture_compression_s3tc (allowing updates at 4x4 boundaries for 2D textures) and NV_texture_compression_vtc (allowing updates at 4x4x4 boundaries for 3D textures) is updated as follows:

"If the internal format of the texture image being modified is COMPRESSED_RGB_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT3_EXT, or COMPRESSED_RGBA_S3TC_DXT5_EXT, the texture is stored using one of several S3TC or VTC compressed texture image formats. Since these algorithms support only 2D and 3D images, CompressedTexSubImage1DARB produces an INVALID_ENUM error if <format> is an S3TC/VTC format. Since S3TC/VTC images are easily edited along 4x4, 4x4x1, or 4x4x4 texel boundaries, the limitations on CompressedTexSubImage2D and CompressedTexSubImage3D are relaxed. CompressedTexSubImage2D and CompressedTexSubImage3D will result in an INVALID_OPERATION error only if one of the following conditions occurs:

```
        * <width> is not a multiple of four or equal to TEXTURE_WIDTH.
        * <height> is not a multiple of four or equal to TEXTURE_HEIGHT.
        * <xoffset> or <yoffset> is not a multiple of four.
        * <depth> is not a multiple of four or equal to TEXTURE_DEPTH, and
          <target> is TEXTURE_3D.
        * <zoffset> is not a multiple of four and <target> is TEXTURE_3D."

   (Note:  The original version of this specification incorrectly failed to
    allow compressed subimage updates of array textures via
    CompressedTexSubImage3D, except at 4x4x4 boundaries/sizes.  This
    undesirable behavior was also implemented by all NVIDIA OpenGL drivers
    published prior to February 2008.)
```

Errors

None. Some error conditions are removed, due to the ability to use the new TEXTURE_1D_ARRAY_EXT and TEXTURE_2D_ARRAY_EXT enums.

New State

(add to table 6.15, p. 276)

Get Value	Type	Get Command	Initial Value	Description	Sec.	Attribute
TEXTURE_BINDING_1D_ARRAY_EXT	2*xZ+	GetIntegerv	0	texture object bound to TEXTURE_1D_ARRAY	3.8.12	texture
TEXTURE_BINDING_2D_ARRAY_EXT	2*xZ+	GetIntegerv	0	texture object bound to TEXTURE_2D_ARRAY	3.8.12	texture

New Implementation Dependent State

(add to Table 6.32, p. 293)

Get Value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
MAX_TEXTURE_ARRAY_LAYERS_EXT	Z+	GetIntegerv	64	maximum number of layers for texture arrays	3.8.1	-

Modifications to The OpenGL Shading Language Specification, Version 1.10.59

(This section describes additions to GLSL to allow shaders to access array textures. This is a subset of the new shading language provided by the EXT_gpu_shader4 extension, limited to array texture support. It is provided here in case implementations choose to support EXT_texture_array without supporting EXT_gpu_shader4 or equivalent functionality.

Note that if the EXT_gpu_shader4 extension is enabled in a shader via an "#extension" line, there is no need to separately enable EXT_texture_array.)

Including the following line in a shader can be used to control the language features described in this extension:

```
  #extension GL_EXT_texture_array : <behavior>
```

where <behavior> is as specified in section 3.3.

A new preprocessor #define is added to the OpenGL Shading Language:

```
  #define GL_EXT_texture_array 1
```

Add to section 3.6 "Keywords"

The following new sampler types are added:

```
  sampler1DArray, sampler2DArray, sampler1DArrayShadow,
  sampler2DArrayShadow
```

Add to section 4.1 "Basic Types"

Add the following entries to the type table:

sampler1DArray	handle for accessing a 1D array texture
sampler2DArray	handle for accessing a 2D array texture
sampler1DArrayShadow	handle for accessing a 1D array depth texture with comparison
sampler2DArrayShadow	handle for accessing a 2D array depth texture with comparison

Add to section 8.7 "Texture Lookup Functions"

Add new functions to the set of allowed texture lookup functions:

Syntax:

```
  vec4 texture1DArray(sampler1DArray sampler, vec2 coord
                      [, float bias])
  vec4 texture1DArrayLod(sampler1DArray sampler, vec2 coord,
                         float lod)
```

Description:

Use the first element (coord.s) of texture coordinate coord to do a texture lookup in the layer indicated by the second coordinate coord.t of the 1D texture array currently bound to sampler. The layer to access is computed by layer = max (0, min(d - 1, floor (coord.t + 0.5)) where 'd' is the depth of the texture array.

Syntax:

```
  vec4 texture2DArray(sampler2DArray sampler, vec3 coord
                        [, float bias])
  vec4 texture2DArrayLod(sampler2DArray sampler, vec3 coord,
                           float lod)
```

Description:

Use the first two elements (coord.s, coord.t) of texture coordinate coord to do a texture lookup in the layer indicated by the third coordinate coord.p of the 2D texture array currently bound to sampler. The layer to access is computed by layer = max (0, min(d - 1, floor (coord.p + 0.5)) where 'd' is the depth of the texture array.

Syntax:

```
  vec4 shadow1DArray(sampler1DArrayShadow sampler, vec3 coord,
                     [float bias])
  vec4 shadow1DArrayLod(sampler1DArrayShadow sampler,
                        vec3 coord, float lod)
```

Description:

Use texture coordinate coord.s to do a depth comparison lookup on an array layer of the depth texture bound to sampler, as described in section 3.8.14 of version 2.0 of the OpenGL specification. The layer to access is indicated by the second coordinate coord.t and is computed by layer = max (0, min(d - 1, floor (coord.t + 0.5)) where 'd' is the depth of the texture array. The third component of coord (coord.p) is used as the R value. The texture bound to sampler must be a depth texture, or results are undefined.

Syntax:

```
  vec4 shadow2DArray(sampler2DArrayShadow sampler, vec4 coord)
```

Description:

Use texture coordinate (coord.s, coord.t) to do a depth comparison lookup on an array layer of the depth texture bound to sampler, as described in section 3.8.14 of version 2.0 of the OpenGL specification. The layer to access is indicated by the third coordinate coord.p and is computed by layer = max (0, min(d - 1, floor (coord.p + 0.5)) where 'd' is the depth of the texture array. The fourth component of coord (coord.q) is used as the R value. The texture bound to sampler must be a depth texture, or results are undefined.

Issues

(1) Should this extension generalize the notion of 1D and 2D textures to be arrays of 1D or 2D images, or simply introduce new targets?

RESOLVED: Introduce new targets.

It would have been possible to simply extend the notion of 1D and 2D textures, and allow applications to pass TEXTURE_1D to TexImage2D (1D arrays) or TEXTURE_2D to TexImage3D (2D arrays). This would have avoided introducing a new set of texture targets (and proxy targets), and a "default texture" (object zero) for each new target.

It is desirable to have a distinction between array and non-array textures in programmable shaders, so compilers can generate code appropriate to the texture type. For "normal" textures, a 2D texture requires two component texture coordinates, while a 2D array texture requires three. Without a distinction between array and non-array textures, implementations must choose between compiling shaders to the most general form (2D arrays) or recompiling shaders based on texture usage. Texture lookups with shadow mapping, LOD bias, or per-pixel LOD have additional complexity, and the interpretation of a coordinate vector may need to depend on whether the texture was an array or non-array texture.

It would be possible to limit the distinction between array and non-array textures to the shaders, but it could then become the responsibility of the application developer to ensure that a texture with multiple layers is used when an "array lookup" is performed, and that a single-layer texture is used when a "non-array lookup" is performed. That begs the question of what the distinction between an "array texture" and a "non-array texture" is. At least two possible distinctions have been identified: one vs. multiple layers, or the API call used to specify the texture (TexImage3D with TEXTURE_2D == array texture, TexImage2D == non-array texture). The former does not allow for the possibility of single-layer array textures; it may be the case that application developers want to use a general shader supporting array textures, but there may be cases where only a single layer might be provided. The latter approach allows for single-layer array textures, but the distinction is now based on the API call.

Adding separate targets eliminates the need for such a distinction. "Array lookups" refer to the TEXTURE_1D_ARRAY_EXT or TEXTURE_2D_ARRAY_EXT targets; "non-array lookups" refer to TEXTURE_1D or TEXTURE_2D. There is never a case where the wrong kind of texture can be used, as TEXTURE_1D_ARRAY_EXT and TEXTURE_2D_ARRAY_EXT textures are always arrays by definition.

This distinction should also be helpful if and when fixed-function fragment processing is supported; the enabled texture target is used to generate an internal fragment shader using the proper "array lookup". There would be no need to recompile shaders depending on whether an enabled texture is an "array texture" or not.

(2) *Should texture arrays be supported for fixed-function fragment processing?*

RESOLVED: No; it's not believed to be worth the effort. Fixed-function fragment processing could be easily supported by allowing applications to enable or disable TEXTURE_1D_ARRAY_EXT or TEXTURE_2D_ARRAY_EXT.

Note that for fixed-function fragment processing, there would be issues with texture lookups of two-dimensional array textures with shadow mapping. Given that all texture lookups are projective, a total of five coordinate components would be required (s, t, layer, depth, q).

(3) *If fixed-function were supported, should the layer number (T or R) be divided by Q in projective texture lookups?*

RESOLVED: It doesn't need to be resolved in this extension, but it would be a problem. There are probably cases where an application would want the divide (handle R more-or-less like S/T); there are probably other cases where the divide would not be wanted. Many developers won't care, and may not even know what the Q coordinate is used for! The default of 1.0 allows applications that don't care about projective lookups to simply ignore that fact.

For programmable fragment shading, an application can code it either way and use non-projective lookups. To the extent that the divide by Q for projective lookups is "free" or "cheap" on OpenGL hardware, compilers may be able to recognize a projective pattern in the computed coordinates and generate code appropriately.

(4) Should DEPTH_COMPONENT textures be supported for texture arrays?

RESOLVED: Yes; multi-layer shadow maps are useful.

(5) How should shadow mapping in texture arrays work with programmable shaders, and fixed-function shaders (if ever supported)?

RESOLVED: The layer number is in the "next" coordinate following the normal 1D or 2D coordinate. That's the "t" coordinate for 1D arrays and the "r" coordinate for 2D arrays. For shadow maps, this is a problem, as the "r" coordinate is generally used as the depth reference value. This is resolved by instead taking the depth reference value from the "q" coordinate.

For some programmable texture lookups (explicit LOD, LOD bias, projective), "too many" coordinates are required. Such lookups are not possible with four-component vectors; it would require at least two parameters to perform such operations.

For fixed-function shading, it is recommended that shadow mapping lookups in two-dimensional array textures be treated as non-projective, even though all other lookups would be projective. Additionally, the "q" coordinate should be used for the depth reference value in this case.

(6) How do texture borders interact with array textures?

RESOLVED: Each individual layer of an array texture can have a border, as though it were a normal one- or two-dimensional texture. However, there are no "border layers".

(7) How does mipmapping work with array textures?

RESOLVED: Level <N+1> is half the size of level <N> in width and/or height, but the number of layers is always the same for each level -- layer <M> of level <N+1> is expected to be a filtered version of layer <M> of the higher mipmap levels. This behavior impacts the texture consistency rules for array textures.

(8) Are compressed textures supported for array textures?

RESOLVED: Yes; they may be loaded via normal TexImage APIs, as well as CompressedTexImage2D and CompressedTexImage3D. Compressed array textures are treated as arrays of compressed 1D or 2D images.

(9) Should these things be called "array textures" or "texture arrays"?

RESOLVED: "Array textures", mostly because it was easier spec wording. Calling them "array textures" also seems like better disambiguation; there are several different things that can be thought of as "texture arrays":

* the array of texture levels (mipmapping)
* the array of texture layers (array textures)
* the array of texels in each image

This spec changes the use of "texture array" in the core specification (which means the array of texels) to instead refer to "texel array".

(10) If they're called "array textures", why does the extension name include "texture_array"?

RESOLVED: Because this is primarily a texture extension, and all such extensions start with "texture".

(11) Should new functions be provided for loading or modifying array textures?

RESOLVED: No. Existing TexImage2D (1D arrays) and TexImage3D (2D arrays), plus corresponding TexSubImage, CopyTexImage, and CopyTexSubImage calls are sufficient.

(12) Should ARB_imaging functionality to be extended to support two-dimensional array textures?

RESOLVED: No. Convolution is rarely used when texture images are defined, and is even less likely for array teture images. This could be addressed via a separate extension if the need were identified, and such operations could be defined for 3D textures as well at that time.

Note that with the API chosen, one-dimensional array textures do have convolution applied (if enabled), because image data is treated as a normal two-dimensional image.

(13) What if an application wants to populate an array texture using separate mipmap chains a layer at a time rather than specifying all layers of a given mipmap level at once?

RESOLVED: For 2D array textures, call TexImage3D once with a NULL image pointer for each level to establish the texel array sizes. Then, call TexSubImage3D for each layer/mipmap level to define individual images.

(14) Should we provide a way to query a single layer of an array texture?

RESOLVED: No; we don't expect this to be an issue in practice. GetTexImage() will return a two- or three-dimensional image for one- and two-dimensional arrays, including all levels. If this were identified as an important need, a follow-on extension could be added in the future.

(15) How is the LOD (lambda) computed for array textures?

RESOLVED: LOD is computed in the same manner for 1D and 2D array textures as it is for normal 1D and 2D textures. The layer coordinate has no effect on LOD computations.

(16) What's the deal with this new "COMPARE_REF_DEPTH_TO_TEXTURE_EXT"?

RESOLVED: It's a new name for the existing enumerant "COMPARE_R_TO_TEXTURE". This alternate name is provided to reflect the fact that it's not always the R coordinate that is used for depth comparisons.

(17) *How do array textures work with framebuffer objects (EXT_framebuffer_object extension, also known as "FBO")?*

RESOLVED: A new function, FramebufferTextureLayerEXT(), is provided to attach a single layer of a one- or two-dimensional array texture to an framebuffer attachment point. That new function can also be used to attach a layer of a three-dimensional texture.

In addition to supporting FBO attachments, the manual mipmap generation support provided by glGenerateMipmapEXT is extended to array textures. Mipmap generation applies to each layer of the array texture independently, as is the case with the GENERATE_MIPMAPS texture parameter.

This support provided here a limited subset of the FBO support added by NV_geometry_program4, which additionally provides the ability to attach an entire level of a three-dimensional, cube map, or array texture. When such attachments are performed, a geometry program can be used to select a layer to render each emitted primitive to.

(18) *Should array texture targets be supported for creation of "render buffers"?*

RESOLVED: No. These are inherently two-dimensional images.

(19) *Should we provide a mipmap generation function to generate mipmaps for only a single layer of an array texture?*

RESOLVED: Not in this extension. We considered adding this toward the end of the development of this extension, but decided not to add it because this mipmap generation function would have very different requirements from the GenerateMipmapEXT function provided by EXT_framebuffer_object.

The existing GenerateMipmapEXT function replaces all levels of detail below the base level with generated mipmaps. If those mipmap levels are unpopulated or inconsistent with the base level, they are completely overwritten with a generated image that is consistent with the base level. If we were to provide a function to generate mipmaps for only a single layer, all other layers of non-base levels would need to be preserved. However, since there are not separate formats or sizes per level, this form of mipmap generation would require that all non-base levels be present and consistent with the base level, or mipmap generation wouldn't work.

We expect that future revisions of the GL will change the specification of mipmapped textures in

(20) *This extension allows the use of S3TC texture internal formats in TexImage3D and CompressedTexImage3D. Does this mean that they are now supported for 3D textures?*

RESOLVED: No. With this extension alone, TexImage3D and CompressedTexImage3D only support S3TC compressed formats with a target of TEXTURE_2D_ARRAY_EXT. The S3TC tokens were added to the list of internal formats supported by TexImage3D and friends because

    two-dimensional array textures are specified using the three-dimensional
    TexImage functions.

    The existing extension NV_texture_compression_vtc does provides support
    for S3TC-style compressed 3D textures.

Revision History

```
    Rev.    Date    Author    Changes
    ----  --------  --------  -----------------------------------------
      6   02/04/08  pbrown    Added a missing interaction with the VTC texture
                              compression spec allowing updates of compressed
                              2D array textures along 4x4x1 boundaries (we
                              previously inherited the VTC restriction of
                              4x4x4).

      5   12/15/06  pbrown    Documented that the '#extension' token
                              for this extension should begin with "GL_",
                              as apparently called for per convention.

      4      --               Pre-release revisions.
```

Name

    EXT_texture_buffer_object

Name Strings

    GL_EXT_texture_buffer_object

Contact

    Pat Brown, NVIDIA Corporation (pbrown 'at' nvidia.com)

Status

    Shipping for GeForce 8 Series (November 2006, Release 95)

Version

```
    Last Modified Date:         10/30/2007
    NVIDIA Revision:            4
```

Number

    330

Dependencies

    OpenGL 2.0 is required.

    NV_gpu_program4 is required.

    This extension is written against the OpenGL 2.0 specification.

    This extension depends trivially on EXT_texture_array.

    This extension depends trivially on NV_texture_shader.

    This extension depends trivially on EXT_texture_integer.

    This extension depends trivially on ARB_texture_float.

    This extension depends trivially on ARB_half_float_pixel.

Overview

    This extension provides a new texture type, called a buffer texture.
    Buffer textures are one-dimensional arrays of texels whose storage comes
    from an attached buffer object. When a buffer object is bound to a buffer
    texture, a format is specified, and the data in the buffer object is
    treated as an array of texels of the specified format.

    The use of a buffer object to provide storage allows the texture data to
    be specified in a number of different ways: via buffer object loads
    (BufferData), direct CPU writes (MapBuffer), framebuffer readbacks
    (EXT_pixel_buffer_object extension). A buffer object can also be loaded
    by transform feedback (NV_transform_feedback extension), which captures
    selected transformed attributes of vertices processed by the GL. Several

of these mechanisms do not require an extra data copy, which would be required when using conventional TexImage-like entry points.

Buffer textures do not support mipmapping, texture lookups with normalized floating-point texture coordinates, and texture filtering of any sort, and may not be used in fixed-function fragment processing. They can be accessed via single texel fetch operations in programmable shaders. For assembly shaders (NV_gpu_program4), the TXF instruction is used. For GLSL, a new sampler type and texel fetch function are used.

While buffer textures can be substantially larger than equivalent one-dimensional textures; the maximum texture size supported for buffer textures in the initial implementation of this extension is 2^27 texels, versus 2^13 (8192) texels for otherwise equivalent one-dimensional textures. When a buffer object is attached to a buffer texture, a size is not specified; rather, the number of texels in the texture is taken by dividing the size of the buffer object by the size of each texel.

New Procedures and Functions

```
void TexBufferEXT(enum target, enum internalformat, uint buffer);
```

New Tokens

Accepted by the <target> parameter of BindBuffer, BufferData, BufferSubData, MapBuffer, BindTexture, UnmapBuffer, GetBufferSubData, GetBufferParameteriv, GetBufferPointerv, and TexBufferEXT, and the <pname> parameter of GetBooleanv, GetDoublev, GetFloatv, and GetIntegerv:

```
    TEXTURE_BUFFER_EXT                              0x8C2A
```

Accepted by the <pname> parameters of GetBooleanv, GetDoublev, GetFloatv, and GetIntegerv:

```
    MAX_TEXTURE_BUFFER_SIZE_EXT                     0x8C2B
    TEXTURE_BINDING_BUFFER_EXT                      0x8C2C
    TEXTURE_BUFFER_DATA_STORE_BINDING_EXT           0x8C2D
    TEXTURE_BUFFER_FORMAT_EXT                       0x8C2E
```

Additions to Chapter 2 of the OpenGL 2.0 Specification (OpenGL Operation)

None.

Additions to Chapter 3 of the OpenGL 2.0 Specification (Rasterization)

(Insert new Section 3.8.4, Buffer Textures. Renumber subsequent sections.)

In addition to one-, two-, and three-dimensional and cube map textures described in previous sections, one additional type of texture is supported. A buffer texture is similar to a one-dimensional texture. However, unlike other texture types, the texel array is not stored as part of the texture. Instead, a buffer object is attached to a buffer texture and the texel array is taken from the data store of an attached buffer object. When the contents of a buffer object's data store are modified, those changes are reflected in the contents of any buffer texture to which the buffer object is attached. Also unlike other textures, buffer textures do not have multiple image levels; only a single data store is available.

The command

```
  void TexBufferEXT(enum target, enum internalformat, uint buffer);
```

attaches the storage for the buffer object named <buffer> to the active buffer texture, and specifies an internal format for the texel array found in the attached buffer object. If <buffer> is zero, any buffer object attached to the buffer texture is detached, and no new buffer object is attached. If <buffer> is non-zero, but is not the name of an existing buffer object, the error INVALID_OPERATION is generated. <target> must be TEXTURE_BUFFER_EXT. <internalformat> specifies the storage format, and must be one of the sized internal formats found in Table X.1.

When a buffer object is attached to a buffer texture, the buffer object's data store is taken as the texture's texel array. The number of texels in the buffer texture's texel array is given by

```
  floor(<buffer_size> / (<components> * sizeof(<base_type>)),
```

where <buffer_size> is the size of the buffer object, in basic machine units and <components> and <base_type> are the element count and base data type for elements, as specified in Table X.1. The number of texels in the texel array is then clamped to the implementation-dependent limit MAX_TEXTURE_BUFFER_SIZE_EXT. When a buffer texture is accessed in a shader, the results of a texel fetch are undefined if the specified texel number is greater than or equal to the clamped number of texels in the texel array.

When a buffer texture is accessed in a shader, an integer is provided to indicate the texel number being accessed. If no buffer object is bound to the buffer texture, the results of the texel access are undefined. Otherwise, the attached buffer object's data store is interpreted as an array of elements of the GL data type corresponding to <internalformat>. Each texel consists of one to four elements that are mapped to texture components (R, G, B, A, L, and I). Element <m> of the texel numbered <n> is taken from element <n> * <components> + <m> of the attached buffer object's data store. Elements and texels are both numbered starting with zero. For texture formats with normalized components, the extracted values are converted to floating-point values according to Table 2.9. The components of the texture are then converted to an (R,G,B,A) vector according to Table X.21, and returned to the shader as a four-component result vector with components of the appropriate data type for the texture's internal format. The base data type, component count, normalized component information, and mapping of data store elements to texture components is specified in Table X.1.

Sized Internal Format	Base Type	Components	Norm	Component 0 1 2 3
ALPHA8	ubyte	1	Y	A . . .
ALPHA16	ushort	1	Y	A . . .
ALPHA16F_ARB	half	1	N	A . . .
ALPHA32F_ARB	float	1	N	A . . .
ALPHA8I_EXT	byte	1	N	A . . .
ALPHA16I_EXT	short	1	N	A . . .
ALPHA32I_EXT	int	1	N	A . . .
ALPHA8UI_EXT	ubyte	1	N	A . . .
ALPHA16UI_EXT	ushort	1	N	A . . .
ALPHA32UI_EXT	uint	1	N	A . . .
LUMINANCE8	ubyte	1	Y	L . . .
LUMINANCE16	ushort	1	Y	L . . .
LUMINANCE16F_ARB	half	1	N	L . . .
LUMINANCE32F_ARB	float	1	N	L . . .
LUMINANCE8I_EXT	byte	1	N	L . . .
LUMINANCE16I_EXT	short	1	N	L . . .
LUMINANCE32I_EXT	int	1	N	L . . .
LUMINANCE8UI_EXT	ubyte	1	N	L . . .
LUMINANCE16UI_EXT	ushort	1	N	L . . .
LUMINANCE32UI_EXT	uint	1	N	L . . .
LUMINANCE8_ALPHA8	ubyte	2	Y	L A . .
LUMINANCE16_ALPHA16	ushort	2	Y	L A . .
LUMINANCE_ALPHA16F_ARB	half	2	N	L A . .
LUMINANCE_ALPHA32F_ARB	float	2	N	L A . .
LUMINANCE_ALPHA8I_EXT	byte	2	N	L A . .
LUMINANCE_ALPHA16I_EXT	short	2	N	L A . .
LUMINANCE_ALPHA32I_EXT	int	2	N	L A . .
LUMINANCE_ALPHA8UI_EXT	ubyte	2	N	L A . .
LUMINANCE_ALPHA16UI_EXT	ushort	2	N	L A . .
LUMINANCE_ALPHA32UI_EXT	uint	2	N	L A . .
INTENSITY8	ubyte	1	Y	I . . .
INTENSITY16	ushort	1	Y	I . . .
INTENSITY16F_ARB	half	1	N	I . . .
INTENSITY32F_ARB	float	1	N	I . . .
INTENSITY8I_EXT	byte	1	N	I . . .
INTENSITY16I_EXT	short	1	N	A . . .
INTENSITY32I_EXT	int	1	N	A . . .
INTENSITY8UI_EXT	ubyte	1	N	A . . .
INTENSITY16UI_EXT	ushort	1	N	A . . .
INTENSITY32UI_EXT	uint	1	N	A . . .
RGBA8	ubyte	4	Y	R G B A
RGBA16	ushort	4	Y	R G B A
RGBA16F_ARB	half	4	N	R G B A
RGBA32F_ARB	float	4	N	R G B A
RGBA8I_EXT	byte	4	N	R G B A
RGBA16I_EXT	short	4	N	R G B A
RGBA32I_EXT	int	4	N	R G B A
RGBA8UI_EXT	ubyte	4	N	R G B A
RGBA16UI_EXT	ushort	4	N	R G B A
RGBA32UI_EXT	uint	4	N	R G B A

Table X.1, Internal Formats for Buffer Textures. For each format, the data type of each element is indicated in the "Base Type" column and the element count is in the "Components" column. The "Norm" column indicates whether components should be treated as normalized floating-point values. The "Component 0, 1, 2, and 3" columns indicate the mapping of each element of a texel to texture components.

In addition to attaching buffer objects to textures, buffer objects can be bound to the buffer object target named TEXTURE_BUFFER_EXT, in order to specify, modify, or read the buffer object's data store. The buffer object bound to TEXTURE_BUFFER_EXT has no effect on rendering. A buffer object is bound to TEXTURE_BUFFER_EXT by calling BindBuffer with <target> set to TEXTURE_BUFFER_EXT. If no corresponding buffer object exists, one is initialized as defined in section 2.9.

The commands BufferData, BufferSubData, MapBuffer, and UnmapBuffer may all be used with <target> set to TEXTURE_BUFFER_EXT. In this case, these commands operate in the same fashion as described in section 2.9, but on the buffer currently bound to the TEXTURE_BUFFER_EXT target.

Modify Section 3.8.11, Texture State and Proxy State (p. 178)

(insert into the first paragraph of the section, p. 178) ... a zero compressed size, and zero-sized components). The buffer texture target contains an integer identifying the buffer object that buffer that provided the data store for the texture, initially zero, and an integer identifying the internal format of the texture, initially LUMINANCE8. Next, there are the two sets of texture properties; ...

Modify Section 3.8.12, Texture Objects (p. 180)

(modify first paragraphs of section, p. 180, simply adding references to buffer textures, which are treated as texture objects)

In addition to the default textures TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP, and TEXTURE_BUFFER_EXT, named one-, two-, and three-dimensional, cube map, and buffer texture objects can be created and operated upon. The name space for texture objects is the unsigned integers, with zero reserved by the GL.

A texture object is created by binding an unused name to TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP, or TEXTURE_BUFFER_EXT. The binding is effected by calling

```
  void BindTexture( enum target, uint texture );
```

with target set to the desired texture target and texture set to the unused name. The resulting texture object is a new state vector, comprising all the state values listed in section 3.8.11, set to the same initial values. If the new texture object is bound to TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP, or TEXTURE_BUFFER_EXT, it is and remains a one-, two-, three-dimensional, cube map, or buffer texture respectively until it is deleted.

BindTexture may also be used to bind an existing texture object to either TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP, or TEXTURE_BUFFER_EXT. The error INVALID_OPERATION is generated if an attempt is made to bind a texture object of different dimensionality than the specified target. If the bind is successful no change is made to the state of the bound texture object, and any previous binding to target is broken.

...

In the initial state, TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP, and TEXTURE_BUFFER_EXT have one-, two-, three-dimensional, cube map, and buffer texture state vectors respectively associated with them. In order that access to these initial textures not be lost, they are treated as texture objects all of whose names are 0. The initial one-, two-, three-dimensional, cube map, and buffer texture is therefore operated upon, queried, and applied as TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP, or TEXTURE_BUFFER_EXT respectively while 0 is bound to the corresponding targets.

Texture objects are deleted by calling

```
  void DeleteTextures( sizei n, uint *textures );
```

textures contains n names of texture objects to be deleted. After a texture object is deleted, it has no contents or dimensionality, and its name is again unused. If a texture that is currently bound to one of the targets TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP, or TEXTURE_BUFFER_EXT is deleted, it is as though BindTexture had been executed with the same target and texture zero. Unused names in textures are silently ignored, as is the value zero.

(modify second paragraph, p. 182, adding buffer textures, plus cube map textures, which is an oversight in the core specification)

The texture object name space, including the initial one-, two-, and three-dimensional, cube map, and buffer texture objects, is shared among all texture units. A texture object may be bound to more than one texture unit simultaneously. After a texture object is bound, any GL operations on that target object affect any other texture units to which the same texture object is bound.

Additions to Chapter 4 of the OpenGL 2.0 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 2.0 Specification (Special Functions)

Modify Section 5.4, Display Lists (p. 237)

(modify "Vertex buffer objects" portion of the list of non-listable commands, p. 241)

  Buffer objects: GenBuffers, DeleteBuffers, BindBuffer, BufferData, BufferSubData, MapBuffer, UnmapBuffer, and TexBufferEXT.

Additions to Chapter 6 of the OpenGL 2.0 Specification (State and State Requests)

Modify Section 6.1.13, Buffer Object Queries (p. 255)

(modify the first paragraph on p. 256) The command

```
  void GetBufferSubData( enum target, intptr offset,
                         sizeiptr size, void *data );
```

    queries the data contents of a buffer object. target is ARRAY_BUFFER,
    ELEMENT_ARRAY_BUFFER, or TEXTURE_BUFFER_EXT. ...

    (modify the last paragraph of the section, p. 256) While the data store of
    a buffer object is mapped, the pointer to the data store can be queried by
    calling

```
      void GetBufferPointerv( enum target, enum pname, void **params );
```

    with target set to ARRAY_BUFFER, ELEMENT_ARRAY_BUFFER, or
    TEXTURE_BUFFER_EXT, and pname set to BUFFER MAP POINTER.

Additions to Appendix A of the OpenGL 2.0 Specification (Invariance)

    None.

Additions to the AGL/GLX/WGL Specifications

    None.

Dependencies on EXT_texture_array

    If EXT_texture_array is supported, the introductory language describing
    buffer textures should acknowledge the existence of array textures. Other
    than that, there are no dependencies between the two extensions.

Dependencies on NV_texture_shader

    If NV_texture_shader is not supported, references to the signed normalized
    internal formats provided by that extension should be removed, and such
    formats may not be passed to TexBufferEXT.

Dependencies on EXT_texture_integer

    If EXT_texture_integer is not supported, references to the signed and
    unsigned integer internal formats provided by that extension should be
    removed, and such formats may not be passed to TexBufferEXT.

Dependencies on ARB_texture_float

    If ARB_texture_float is not supported, references to the floating-point
    internal formats provided by that extension should be removed, and such
    formats may not be passed to TexBufferEXT.

Dependencies on ARB_half_float_pixel

    If ARB_texture_float is not supported, references to the 16-bit
    floating-point internal formats provided by ARB_texture_float should be
    removed, and such formats may not be passed to TexBufferEXT. If an
    implementation supports ARB_texture_float, but does not support
    ARB_half_float_pixel, 16-bit floating-point texture formats may be
    available using normal texture mechanisms, but not with buffer textures.

Errors

    INVALID_OPERATION is generated by TexBufferEXT if <buffer> is non-zero and
    is not the name of an existing buffer object.

New State

(add to table 6.15, Texture State Per Texture Unit/Binding Point p. 276)

Get Value	Type	Get Command	Initial Value	Description	Sec.	Attribute
TEXTURE_BINDING_BUFFER_EXT	2*xZ+	GetIntegerv	0	Texture object bound to TEXTURE_BUFFER_EXT	3.8.12	texture

(add to table 6.16, Texture State Per Texture Object, p. 276)

Get Value	Type	Get Command	Initial Value	Description	Sec.	Attribute
TEXTURE_BUFFER_DATA_STORE_ BINDING_EXT	nxZ+	GetIntegerv	0	Buffer object bound as the data store for the active image unit's buffer texture	3.8.12	texture
TEXTURE_BUFFER_FORMAT_EXT	nxZ+	GetIntegerv	LUMIN- ANCE8	Internal format for the active image unit's buffer texture	3.8.12	texture

(add to table 6.37, Miscellaneous State, p. 298)

Get Value	Type	Get Command	Initial Value	Description	Sec.	Attribute
TEXTURE_BUFFER_EXT	Z+	GetIntegerv	0	Buffer object bound to the generic buffer texture binding point	3.8.12	texture

New Implementation Dependent State

(modify Table 6.32, p. 293)

Get Value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
MAX_TEXTURE_BUFFER_SIZE_EXT	Z+	GetIntegerv	65536	number of addressable texels for buffer textures	3.8.4	-

Issues

(1) *Buffer textures are potentially large one-dimensional arrays that can be accessed with single-texel fetches. How should this functionality be exposed?*

RESOLVED: Several options were considered. The final approach creates a new type of texture object, called a buffer texture, whose texel array is taken from the data store from a buffer object. The combined set of extensions using buffer objects provides numerous locations where the GL can read and write data to a buffer object:

EXT_vertex_buffer_object allows vertex attributes to be pulled from a buffer object.

EXT_pixel_buffer_object allows pixel operations (DrawPixels, ReadPixels, TexImage) to read or write data to a buffer object.

EXT_parameter_buffer_object and EXT_bindable_uniform allows assembly vertex, fragment, and geometry programs, and all GLSL shaders to read program parameter / uniform data from a buffer object.

EXT_texture_buffer_object allows programs to read texture data from a buffer object.

NV_transform_feedback allows programs to write transformed vertex attributes to a buffer object.

When combined, interesting feedback paths are possible, where large arrays of data can be generated by the GPU and the consumed by it in multi-pass algorithms, using the buffer object's storage to hold intermediate data. This allows applications to run complicated algorithms on the GPU without necessarily pulling data back to host CPU for additional processing.

Given that buffer object memory is visible to users as raw memory, all uses of the memory must have well-defined data formats. For VBO and PBO, those formats are explicitly given by calls such as VertexPointer, TexImage2D, or ReadPixels. When used as a buffer texture, it is necessary to specify an internal format with which the bytes of the buffer object's data store are interpreted.

Another option considered was to greatly increase the maximum texture size for 1D texture. This has the advantage of not requiring new mechanisms. However, there are a couple limitations of this approach. First, conventional textures have their own storage that is not accessible elsewhere, which limits some of the sharing opportunities described above. Second, buffer textures do have slightly different hardware implementations than 1D textures. In the hardware of interest, "normal" 1D textures can be mipmapped and filtered, but have a maximum size that is considerably smaller than that supported for buffer textures. If both texture types used the same API mechanism, it might be necessary to reprogram texture hardware and/or shaders depending on the size of the textures used. This will incur CPU overhead to determine if such reprogramming is necessary and to perform the reprogramming if so.

(2) Since buffer textures borrow storage from buffer objects, whose storage is visible to applications, a format must be imposed on the bytes of the buffer object. What texture formats are supported for buffer objects?

RESOLVED: All sized one-, two-, and four-component internal formats with 8-, 16-, and 32-bit components are supported. Unsized internal formats, and sized formats with other component sizes are also not supported. Three-component (RGB) formats are not supported due to hardware limitations.

All component data types supported for normal textures are also supported for buffer textures. This includes unsigned [0,1] normalized components (e.g., RGBA8), floating-point components from ARB_texture_float (e.g., RGBA32F_ARB), signed and unsigned integer components from EXT_texture_integer (e.g., RGBA8I_EXT, RGBA16UI_EXT), and signed [-1,+1] normalized components from NV_texture_shader (e.g., SIGNED_RGBA8_NV).

(3) How can arrays of three-component vectors be accessed by applications?

RESOLVED: Several approaches are possible.

First, the vectors can be padded out to four components (RGBA), with an extra unused component for each texel. This has a couple undesirable properties: it adds 33% to the required storage and adding the extra component may require reformatting of original data generated by the application. However, the data in this format can be retrieved with a single 32-, 64-, or 128-bit lookup.

Alternately, the buffer texture can be defined using a single component, and a shader can perform three lookups to separately fetch texels 3*N, 3*N+1, and 3*N+2, combining the result in a three-component vector representing "RGB" texel N. This doesn't require extra storage or reformatting and doesn't require additional bandwidth for texture fetches. But it does require additional shader instructions to obtain each texel.

(4) Does this extension support fixed-function fragment processing, somehow allowing buffer textures to be accessed without programmable shaders?

RESOLVED: No. We expect that it would be difficult to properly access a buffer texture and combine the returned texel with other color or texture data, given the extremely limited programming model provided by fixed-function fragment processing.

Note also that the single-precision floating-point representation commonly used by current graphics hardware is not sufficiently precise to exactly represent all texels in a large buffer texture. For example, it is not possible to represent 2^24+1 using the 32-bit IEEE floating-point representation.

(5) What happens if a buffer object is deleted or respecified when bound to a buffer texture?

RESOLVED: BufferData is allowed to be used to update a buffer object that has already been bound to a texture with TexBuffer. The update to the data is not guaranteed to affect the texture until next time it is bound to a texture image unit. When DeleteBuffers is called, any buffer that is bound to a texture is removed from the names array, but remains as long as it is bound to a texture. The buffer is fully removed when the texture unbinds it or when the texture buffer object is deleted.

(6) Should applications be able to modify the data store of a buffer object while it is bound to a buffer texture?

RESOLVED: An application is allowed to update the data store for a buffer object when the buffer object is bound to a texture.

(7) Do buffer textures support texture parameters (TexParameter) or queries (GetTexParameter, GetTexLevelParameter, GetTexImage)?

RESOLVED: No. None of the existing parameters apply to buffer textures, and this extension doesn't introduce the need for any new ones. Buffer textures have no levels, and the size in texels is implicit (based on the data store). Given that the texels themselves are obtained from a buffer object, it seems more appropriate to retrieve such data with buffer object queries. The only "parameter" of a buffer texture is the internal format, which is specified at the same time the buffer object is bound.

Note that the spec edits above don't add explicit error language for any of these cases. That is because each of the functions enumerate the set of valid <target> parameters. Not editing the spec to allow TEXTURE_BUFFER_EXT in these cases means that target is not legal, and an INVALID_ENUM error should be generated.

(8) What about indirect rendering with a mix of big- and little-endian clients? If components are 16- or 32-bit, how are they interpreted?

RESOLVED: Buffer object data are interpreted according to the native representation of the server. If the server and client have different endianness, applications must perform byte swapping as needed to match the server's representation. No mechanism is provided to perform this byte swapping on buffer object updates or when texels are fetched.

The same problem also exists when buffer objects are used for vertex arrays (VBO). For buffer objects used for pixel packing and unpacking (ARB_pixel_buffer_object), the PixelStore byte swapping parameters (PACK_SWAP_BYTES, UNPACK_SWAP_BYTES) would presumably apply and could be used to perform the necessary byte swapping.

(9) Should the set of formats supported for buffer textures be enumerated, or should the extension instead nominally support all formats, but accept only an implementation-dependent subset?

RESOLVED: Provide a specified set of supported formats. This extension simply enumerates all 8-, 16-, and 32-byte internal formats with 1, 2, or 4 components, and specifies the mapping of unformatted buffer object data to texture components. A follow-on extension could be done to support 3-component texels when better native hardware support is available.

Other than 3-component texels, the set of formats supported seems pretty compehensive. We expect that buffer textures would be used for general computational tasks, where there is little need for formats with smaller components (e.g., RGBA4444). Such formats are generally not supported natively on CPUs today. With the general computational model provided by NV_gpu_program4 and EXT_gpu_shader4, it would be possible to treat such "packed" formats as larger single-component formats and unpack them with a small number of shader instructions.

If and when double-precision floats or 64-bit integers are supported as basic types usable by shaders, we would expect that an extension would add new texture internal formats with 64-bit components and that those formats would also be supported for general-purpose textures and buffer textures as well.

(10) How are buffer textures supported in GLSL?

RESOLVED: Create a new sampler type (samplerBuffer) for buffer textures and add a new lookup function (texelFetchBuffer) to explicitly access them using texture hardware.

Other possibilities considered included extending the notion of bindable uniforms to support uniforms whose corresponding buffer objects can be bound to texture resources (e.g., "texture bindable uniform" instead of "bindable uniform"). We also considered automatically assigning bindable uniforms to texture or shader resources as appropriate. Note that the restrictions, size limits, and performance characterstics of buffer textures and parameter buffers (NV_parameter_buffer_object) differ. Automatic handling of uniforms adds driver complexity and may tend to hide performance characteristics since it isn't clear what resource would be used for what variable. Additionally, it could require shader recompilation if the size of a uniform array is variable, and the hardware resource used depended on the size.

In the end, the texture approach seemed the simplest, and we chose that. It might be worth doing something more complex in the future.

(11) What is the TEXTURE_BUFFER_EXT buffer object binding point good for?

RESOLVED: It can be used for loading data into buffer objects, and for mapping and unmapping buffers, both without disturbing other binding points. Otherwise, it has no effect on GL operations, since buffer objects are bound to textures using the TexBufferEXT() command that does not affect the buffer object binding point.

Buffer object binding points have mixed usage. In the EXT_vertex_buffer_object extension (OpenGL 1.5), there are two binding points. The ELEMENT_ARRAY_BUFFER has a direct effect on rendering, as it modifies DrawElements() calls. The effect of ARRAY_BUFFER is much more indirect; it is only used to affect subsequent vertex array calls (e.g., VertexPointer) and has no direct effect on rendering. The reason for this is that the API was retrofitted on top of existing vertex array APIs. If a new vertex array API were created that emphasized or even required the use of buffer objects, it seems likely that the buffer object would be included in the calls equivalent to today's VertexPointer() call.

(12) How is the various buffer texture-related state queried?

RESOLVED: There are three pieces of state that can be queried: (a) the texture object bound to buffer texture binding point for the active texture image unit, (b) the buffer object whose data store was used by that texture object, and (c) the buffer object bound to the TEXTURE_BUFFER_EXT binding point.

All three are queried with GetIntegerv, because it didn't seem worth the trouble to add one or more new query functions. Note that for (a) and (b), the texture queried is the one bound to TEXTURE_BUFFER_EXT on the active texture image unit.

(13) Should we provide a new set of names for the signed normalized textures introduced in NV_texture_shader that match the convention used for floating-point and integer textures?

RESOLVED: No.

(14) Can a buffer object be attached to more than one buffer texture at once?

RESOLVED: Multiple buffer textures may attach to the same buffer object simultaneously.

(15) How does this extension interact with display lists?

RESOLVED: Buffer object commands can't be compiled into a display list. The new command in this extension uses buffer objects, so we specify that it also can't be compiled into a display list.

Revision History

```
    Rev.    Date    Author    Changes
    ----  --------  --------  -----------------------------------------
      4   10/30/07  ewerness  Add resolutions to various issues

      3      --               Pre-release revisions.
```

Name

    EXT_texture_compression_latc

Name Strings

    GL_EXT_texture_compression_latc
    GL_NV_texture_compression_latc (legacy)

Contributors

    Mark J. Kilgard, NVIDIA
    Pat Brown, NVIDIA
    Yanjun Zhang, S3
    Attila Barsi, Holografika

Contact

    Mark J. Kilgard, NVIDIA Corporation (mjk 'at' nvidia.com)

Status

    Shipping for GeForce 8 Series (November 2006)

Version

    Last Modified Date: 1/21/2008
    Revision: 1.2

Number

    331

Dependencies

    OpenGL 1.3 or ARB_texture_compression required

    This extension is written against the OpenGL 2.0 (September 7,
    2004) specification.

Overview

    This extension introduces four new block-based texture compression
    formats suited for unsigned and signed luminance and luminance-alpha
    textures (hence the name "latc" for Luminance-Alpha Texture
    Compression).

    These formats are designed to reduce the storage requirements and
    memory bandwidth required for luminance and luminance-alpha textures
    by a factor of 2-to-1 over conventional uncompressed luminance and
    luminance-alpha textures with 8-bit components (GL_LUMINANCE8 and
    GL_LUMINANCE8_ALPHA8).

    The compressed signed luminance-alpha format is reasonably suited
    for storing compressed normal maps.

New Procedures and Functions

None.

New Tokens

Accepted by the <internalformat> parameter of TexImage2D, CopyTexImage2D, and CompressedTexImage2D and the <format> parameter of CompressedTexSubImage2D:

COMPRESSED_LUMINANCE_LATC1_EXT	0x8C70
COMPRESSED_SIGNED_LUMINANCE_LATC1_EXT	0x8C71
COMPRESSED_LUMINANCE_ALPHA_LATC2_EXT	0x8C72
COMPRESSED_SIGNED_LUMINANCE_ALPHA_LATC2_EXT	0x8C73

Additions to Chapter 2 of the OpenGL 2.0 Specification (OpenGL Operation)

None.

Additions to Chapter 3 of the OpenGL 2.0 Specification (Rasterization)

-- Section 3.8.1, Texture Image Specification

Add to Table 3.17 (page 155): Specific compressed internal formats

Compressed Internal Format	Base Internal Format
COMPRESSED_LUMINANCE_LATC1_EXT	LUMINANCE
COMPRESSED_SIGNED_LUMINANCE_LATC1_EXT	LUMINANCE
COMPRESSED_LUMINANCE_ALPHA_LATC2_EXT	LUMINANCE_ALPHA
COMPRESSED_SIGNED_LUMINANCE_ALPHA_LATC2_EXT	LUMINANCE_ALPHA

-- Section 3.8.2, Alternative Texture Image Specification Commands

Add to the end of the section (page 163):

"If the internal format of the texture image being modified is COMPRESSED_LUMINANCE_LATC1_EXT, COMPRESSED_SIGNED_LUMINANCE_LATC1_EXT, COMPRESSED_LUMINANCE_ALPHA_LATC2_EXT, or COMPRESSED_SIGNED_LUMINANCE_ALPHA_LATC2_EXT, the texture is stored using one of the two LATC compressed texture image encodings (see appendix). Such images are easily edited along 4x4 texel boundaries, so the limitations on TexSubImage2D or CopyTexSubImage2D parameters are relaxed. TexSubImage2D and CopyTexSubImage2D will result in an INVALID_OPERATION error only if one of the following conditions occurs:

* <width> is not a multiple of four or equal to TEXTURE_WIDTH, unless <xoffset> and <yoffset> are both zero.
* <height> is not a multiple of four or equal to TEXTURE_HEIGHT, unless <xoffset> and <yoffset> are both zero.
* <xoffset> or <yoffset> is not a multiple of four.

The contents of any 4x4 block of texels of an LATC compressed texture image that does not intersect the area being modified are preserved during valid TexSubImage2D and CopyTexSubImage2D calls."

-- Section 3.8.3, Compressed Texture Images

Add after the 4th paragraph (page 164) at the end of the CompressedTexImage discussion:

"If <internalformat> is COMPRESSED_LUMINANCE_LATC1_EXT, COMPRESSED_SIGNED_LUMINANCE_LATC1_EXT, COMPRESSED_LUMINANCE_ALPHA_LATC2_EXT, or COMPRESSED_SIGNED_LUMINANCE_ALPHA_LATC2_EXT, the compressed texture is stored using one of several LATC compressed texture image formats. The LATC texture compression algorithm supports only 2D images without borders. CompressedTexImage1D and CompressedTexImage3D produce an INVALID_ENUM error if <internalformat> is an LATC format. CompressedTexImage2D will produce an INVALID_OPERATION error if <border> is non-zero.

Add to the end of the section (page 166) at the end of the CompressedTexSubImage discussion:

"If the internal format of the texture image being modified is COMPRESSED_LUMINANCE_LATC1_EXT, COMPRESSED_SIGNED_LUMINANCE_LATC1_EXT, COMPRESSED_LUMINANCE_ALPHA_LATC2_EXT, or COMPRESSED_SIGNED_LUMINANCE_ALPHA_LATC2_EXT, the texture is stored using one of the several LATC compressed texture image formats. Since the LATC texture compression algorithm supports only 2D images, CompressedTexSubImage1D and CompressedTexSubImage3D produce an INVALID_ENUM error if <format> is an LATC format. Since LATC images are easily edited along 4x4 texel boundaries, the limitations on CompressedTexSubImage2D are relaxed. CompressedTexSubImage2D will result in an INVALID_OPERATION error only if one of the following conditions occurs:

* <width> is not a multiple of four or equal to TEXTURE_WIDTH.
* <height> is not a multiple of four or equal to TEXTURE_HEIGHT.
* <xoffset> or <yoffset> is not a multiple of four.

The contents of any 4x4 block of texels of an LATC compressed texture image that does not intersect the area being modified are preserved during valid TexSubImage2D and CopyTexSubImage2D calls."

Additions to Chapter 4 of the OpenGL 2.0 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 2.0 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 2.0 Specification (State and State Requests)

None.

Additions to Appendix A of the OpenGL 2.0 Specification (Invariance)

None.

Additions to the AGL/GLX/WGL Specifications

None.

GLX Protocol

None.

Dependencies on ARB_texture_compression

If ARB_texture_compression is supported, all the errors and accepted tokens for CompressedTexImage1D, CompressedTexImage2D, CompressedTexImage3D, CompressedTexSubImage1D, CompressedTexSubImage2D, and CompressedTexSubImage3D also apply respectively to the ARB-suffixed CompressedTexImage1DARB, CompressedTexImage2DARB, CompressedTexImage3DARB, CompressedTexSubImage1DARB, CompressedTexSubImage2DARB, and CompressedTexSubImage3DARB.

Errors

INVALID_ENUM is generated by CompressedTexImage1D or CompressedTexImage3D if <internalformat> is COMPRESSED_LUMINANCE_LACT1_EXT, COMPRESSED_SIGNED_LUMINANCE_LATC1_EXT, COMPRESSED_LUMINANCE_ALPHA_LATC2_EXT, or COMPRESSED_SIGNED_LUMINANCE_ALPHA_LATC2_EXT.

INVALID_OPERATION is generated by CompressedTexImage2D if <internalformat> is COMPRESSED_LUMINANCE_LACT1_EXT, COMPRESSED_SIGNED_LUMINANCE_LATC1_EXT, COMPRESSED_LUMINANCE_ALPHA_LATC2_EXT, or COMPRESSED_SIGNED_LUMINANCE_ALPHA_LATC2_EXT and <border> is not equal to zero.

INVALID_ENUM is generated by CompressedTexSubImage1D or CompressedTexSubImage3D if <format> is COMPRESSED_LUMINANCE_LACT1_EXT, COMPRESSED_SIGNED_LUMINANCE_LATC1_EXT, COMPRESSED_LUMINANCE_ALPHA_LATC2_EXT, or COMPRESSED_SIGNED_LUMINANCE_ALPHA_LATC2_EXT.

INVALID_OPERATION is generated by TexSubImage2D CopyTexSubImage2D, or CompressedTexSubImage2D if TEXTURE_INTERNAL_FORMAT is COMPRESSED_LUMINANCE_LACT1_EXT, COMPRESSED_SIGNED_LUMINANCE_LATC1_EXT, COMPRESSED_LUMINANCE_ALPHA_LATC2_EXT, or COMPRESSED_SIGNED_LUMINANCE_ALPHA_LATC2_EXT and any of the following apply: <width> is not a multiple of four or equal to TEXTURE_WIDTH; <height> is not a multiple of four or equal to TEXTURE_HEIGHT; <xoffset> or <yoffset> is not a multiple of four.

The following restrictions from the ARB_texture_compression specification do not apply to LATC texture formats, since subimage modification is straightforward as long as the subimage is properly aligned.

```
    DELETE: INVALID_OPERATION is generated by TexSubImage1D, TexSubImage2D,
    DELETE: TexSubImage3D, CopyTexSubImage1D, CopyTexSubImage2D, or
    DELETE: CopyTexSubImage3D if the internal format of the texture image is
    DELETE: compressed and <xoffset>, <yoffset>, or <zoffset> does not equal
    DELETE: -b, where b is value of TEXTURE_BORDER.

    DELETE: INVALID_VALUE is generated by CompressedTexSubImage1D,
    DELETE: CompressedTexSubImage2D, or CompressedTexSubImage3D if the
    DELETE: entire texture image is not being edited:  if <xoffset>,
    DELETE: <yoffset>, or <zoffset> is greater than -b, <xoffset> + <width> is
    DELETE: less than w+b, <yoffset> + <height> is less than h+b, or <zoffset>
    DELETE: + <depth> is less than d+b, where b is the value of
    DELETE: TEXTURE_BORDER, w is the value of TEXTURE_WIDTH, h is the value of
    DELETE: TEXTURE_HEIGHT, and d is the value of TEXTURE_DEPTH.
```

See also errors in the GL_ARB_texture_compression specification.

New State

4 new state values are added for the per-texture object GL_TEXTURE_INTERNAL_FORMAT state.

In the "Textures" state table(page 278), increment the TEXTURE_INTERNAL_FORMAT subscript for Z by 4 in the "Type" row.

[NOTE: The OpenGL 2.0 specification actually should read "n x Z48*" because of the 6 generic compressed internal formats in table 3.18.]

New Implementation Dependent State

None

Appendix

LATC Compressed Texture Image Formats

Compressed texture images stored using the LATC compressed image encodings are represented as a collection of 4x4 texel blocks, where each block contains 64 or 128 bits of texel data. The image is encoded as a normal 2D raster image in which each 4x4 block is treated as a single pixel. If an LATC image has a width or height less than four, the data corresponding to texels outside the image are irrelevant and undefined.

When an LATC image with a width of <w>, height of <h>, and block size of <blocksize> (8 or 16 bytes) is decoded, the corresponding image size (in bytes) is:

```
        ceil(<w>/4) * ceil(<h>/4) * blocksize.
```

When decoding an LATC image, the block containing the texel at offset (<x>, <y>) begins at an offset (in bytes) relative to the base of the image of:

```
        blocksize * (ceil(<w>/4) * floor(<y>/4) + floor(<x>/4)).
```

The data corresponding to a specific texel (<x>, <y>) are extracted from a 4x4 texel block using a relative (x,y) value of

```
    (<x> modulo 4, <y> modulo 4).
```

There are four distinct LATC image formats:

COMPRESSED_LUMINANCE_LATC1: Each 4x4 block of texels consists of 64 bits of unsigned luminance image data.

Each luminance image data block is encoded as a sequence of 8 bytes, called (in order of increasing address):

```
        lum0, lum1, bits_0, bits_1, bits_2, bits_3, bits_4, bits_5
```

The 6 "bits_*" bytes of the block are decoded into a 48-bit bit vector:

```
    bits   = bits_0 +
             256 * (bits_1 +
                    256 * (bits_2 +
                           256 * (bits_3 +
                                  256 * (bits_4 +
                                         256 * bits_5))))
```

lum0 and lum1 are 8-bit unsigned integers that are unpacked to luminance values LUM0 and LUM1 as though they were pixels with a <format> of LUMINANCE and a type of UNSIGNED_BTYE.

bits is a 48-bit unsigned integer, from which a three-bit control code is extracted for a texel at location (x,y) in the block using:

```
    code(x,y) = bits[3*(4*y+x)+2..3*(4*y+x)+0]
```

where bit 47 is the most significant and bit 0 is the least significant bit.

```
    The luminance value L for a texel at location (x,y) in the block
    is given by:

        LUM0,              if lum0 > lum1 and code(x,y) == 0
        LUM1,              if lum0 > lum1 and code(x,y) == 1
        (6*LUM0+  LUM1)/7, if lum0 > lum1 and code(x,y) == 2
        (5*LUM0+2*LUM1)/7, if lum0 > lum1 and code(x,y) == 3
        (4*LUM0+3*LUM1)/7, if lum0 > lum1 and code(x,y) == 4
        (3*LUM0+4*LUM1)/7, if lum0 > lum1 and code(x,y) == 5
        (2*LUM0+5*LUM1)/7, if lum0 > lum1 and code(x,y) == 6
        (  LUM0+6*LUM1)/7, if lum0 > lum1 and code(x,y) == 7

        LUM0,              if lum0 <= lum1 and code(x,y) == 0
        LUM1,              if lum0 <= lum1 and code(x,y) == 1
        (4*LUM0+  LUM1)/5, if lum0 <= lum1 and code(x,y) == 2
        (3*LUM0+2*LUM1)/5, if lum0 <= lum1 and code(x,y) == 3
        (2*LUM0+3*LUM1)/5, if lum0 <= lum1 and code(x,y) == 4
        (  LUM0+4*LUM1)/5, if lum0 <= lum1 and code(x,y) == 5
        MINLUM,            if lum0 <= lum1 and code(x,y) == 6
        MAXLUM,            if lum0 <= lum1 and code(x,y) == 7

    MINLUM and MAXLUM are 0.0 and 1.0 respectively.

Since the decoded texel has a luminance format, the resulting RGBA
value for the texel is (L,L,L,1).
```

COMPRESSED_SIGNED_LUMINANCE_LATC1: Each 4x4 block of texels consists of 64 bits of signed luminance image data. The luminance values of a texel are extracted in the same way as COMPRESSED_LUMINANCE_LATC1 except lum0, lum1, LUM0, LUM1, MINLUM, and MAXLUM are signed values defined as follows:

```
    lum0 and lum1 are 8-bit signed (two's complement) integers.

           { lum0 / 127.0, lum0 > -128
    LUM0 = {
           { -1.0,         lum0 == -128

           { lum1 / 127.0, lum1 > -128
    LUM1 = {
           { -1.0,         lum1 == -128

    MINLUM = -1.0

    MAXLUM =  1.0
```

CAVEAT for signed lum0 and lum1 values: the expressions "lum0 > lum1" and "lum0 <= lum1" above are considered undefined (read: may vary by implementation) when lum0 equals -127 and lum1 equals -128, This is because if lum0 were remapped to -127 prior to the comparison to reduce the latency of a hardware decompressor, the expressions would reverse their logic. Encoders for the signed LA formats should avoid encoding blocks where lum0 equals -127 and lum1 equals -128.

COMPRESSED_LUMINANCE_ALPHA_LATC2: Each 4x4 block of texels consists of 64 bits of compressed unsigned luminance image data followed by 64 bits of compressed unsigned alpha image data.

The first 64 bits of compressed luminance are decoded exactly like COMPRESSED_LUMINANCE_LATC1 above.

The second 64 bits of compressed alpha are decoded exactly like COMPRESSED_LUMINANCE_LATC1 above except the decoded value L for this second block is considered the resulting alpha value A.

Since the decoded texel has a luminance-alpha format, the resulting RGBA value for the texel is (L,L,L,A).

COMPRESSED_SIGNED_LUMINANCE_ALPHA_LATC2: Each 4x4 block of texels consists of 64 bits of compressed signed luminance image data followed by 64 bits of compressed signed alpha image data.

The first 64 bits of compressed luminance are decoded exactly like COMPRESSED_SIGNED_LUMINANCE_LATC1 above.

The second 64 bits of compressed alpha are decoded exactly like COMPRESSED_SIGNED_LUMINANCE_LATC1 above except the decoded value L for this second block is considered the resulting alpha value A.

Since this image has a luminance-alpha format, the resulting RGBA value is (L,L,L,A).

Issues

1) *What should these new formats be called?*

RESOLVED: "latc" for Luminance-Alpha Texture Compression.

2) *How should the uncompressed and filtered texels be returned by texture fetches?*

RESOLVED: Luminance values show up as they do conventionally as (L,L,L,1) where the luminance value L is replicated in the red, green, and blue components and alpha is forced to 1. Likewise, luminance-alpha values show up as (L,L,L,A) where A is the alpha value.

Alternatively, prior extensions such as NV_float_buffer and NV_texture_shader have introduced formats such as GL_FLOAT_R_NV and GL_DSDT_NV where one- and two-component texture formats show up as (X,0,0,1) or (X,Y,0,1) RGBA texels. Such formats have not proven popular. In particular, they interact awkwardly with the pixel path and conventional texture environment modes.

The (X,Y,0,1) convention, particularly with signed components, is nice for normal maps because a normalized vector can be formed by a shader program by computing sqrt(abs(1-X*X-Y*Y)) for the Z component. However, this niceness is mostly conceptual however since the same effect can be accomplished with swizzling as shown in this GLSL code:

```
        vec2 texLA  = texture2D(samplerLA, gl_TexCoord[0]).xw;
        vec3 normal = vec3(texLA.x,
                           texLA.y,
                           sqrt(abs(1-texLA.x*texLA.x-texLA.y*texLA.y)));
```

The most important reason to make these new compressed formats show up identically to conventional luminance and luminance-alpha texels is to allow applications to seamlessly substitute the new compressed formats for existing GL_LUMINANCE and GL_LUMINANCE_ALPHA textures. Alternative component arrangements would make it more cumbersome for existing applications to switch over luminance and luminance-alpha textures to these compressed formats.

3) *Should luminance and luminance-alpha compression formats with signed components be introduced when the core specification lacked uncompressed luminance and luminance-alpha texture formats?*

RESOLVED: Yes, signed luminance and luminance-alpha compression formats should be added.

Signed luminance-alpha formats are suited for compressed normal maps. Compressed normal maps may well be the dominant use of this extension.

Unsigned luminance-alpha formats require an extra "expand normal" operation to convert [0,1] to [-1,+1]. Direct support for signed luminance-alpha formats avoids this step in a shader program.

4) *Should there be a mix of signed luminance and unsigned alpha or vice versa?*

RESOLVED: No.

NV_texture_shader provided an internal format (GL_SIGNED_RGB_UNSIGNED_ALPHA_NV) with mixed signed and unsigned components. The format saw little usage. There's no reason to think a GL_SIGNED_LUMINANCE_UNSIGNED_ALPHA format would be any more useful or popular.

5) *How are signed integer values mapped to floating-point values?*

RESOLVED: A signed 8-bit two's complement value X is computed to a floating-point value Xf with the formula:

```
             { X / 127.0, X > -128
        Xf = {
             { -1.0,      X == -128
```

This conversion means -1, 0, and +1 are all exactly representable, however -128 and -127 both map to -1.0. Mapping -128 to -1.0 avoids the numerical awkwardness of have a representable value slightly more negative than -1.0.

This conversion is intentionally NOT the "byte" conversion listed in Table 2.9 for component conversions. That conversion says:

```
Xf = (2*X + 1) / 255.0
```

The Table 2.9 conversion is incapable of exactly representing zero.

6) *How will signed components resulting from GL_COMPRESSED_SIGNED_LUMINANCE_LATC1_EXT and GL_COMPRESSED_SIGNED_LUMINANCE_ALPHA_LATC2_EXT texture fetches interact with fragment coloring?*

RESOLVED: The specification language for this extension is silent about clamping behavior leaving this to the core specification and other extensions. The clamping or lack of clamping is left to the core specification and other extensions.

For assembly program extensions supporting texture fetches (ARB_fragment_program, EXT_fragment_program, EXT_vertex_program3, etc.) or the OpenGL Shading Language, these signed formats will appear as expected with unclamped signed components as a result of a texture fetch instruction.

If ARB_color_buffer_float is supported, its clamping controls will apply.

NV_texture_shader extension, if supported, adds support for fixed-point textures with signed components and relaxed the fixed-function texture environment clamping appropriately. If the NV_texture_shader extension is supported, its specified behavior for the texture environment applies where intermediate values are clamped to [-1,1] unless stated otherwise as in the case of explicitly clamped to [0,1] for GL_COMBINE. or clamping the linear interpolation weight to [0,1] for GL_DECAL and GL_BLEND.

Otherwise, the conventional core texture environment clamps incoming, intermediate, and output color components to [0,1].

This implies that the conventional texture environment functionality of unextended OpenGL 1.5 or OpenGL 2.0 without using GLSL (and with none of the extensions referred to above) is unable to make proper use of the signed texture formats added by this extension because the conventional texture environment requires texture source colors to be clamped to [0,1]. Texture filtering of these signed formats would be still signed, but negative values generated post-filtering would be clamped to zero by the core texture environment functionality. The expectation is clearly that this extension would be co-implemented with one of the previously referred to extensions or used with GLSL for the new signed formats to be useful.

7) *Should a specific normal map compression format be added?*

RESOLVED: No.

It's probably short-sighted to design a format just for normal maps. Indeed, NV_texture_shader added a GL_SIGNED_HILO_NV format with exactly the kind of "hemisphere remap" useful for normal maps and the format went basically unused. Instead, this extension provides the mechanism for compressed normal maps based on the more conventional luminance-alpha format.

The GL_COMPRESSED_LUMINANCE_ALPHA_LATC2_EXT and GL_COMPRESSED_SIGNED_LUMINANCE_ALPHA_LATC2_EXT formats are sufficient for normal maps with additional shader instructions used to generate the 3rd component.

8) *Should uncompressed signed luminance and luminance-alpha formats be added by this extension?*

RESOLVED: No, this extension is focused on just adding compressed texture formats.

The NV_texture_shader extension adds such uncompressed signed texture formats. A distinct multi-vendor extension for signed fixed-point texture formats could provide all or a subset of the signed fixed-point uncompressed texture formats introduced by NV_texture_shader.

9) *What compression ratios does this extension provide?*

The LATC1 formats are 8 bytes (64 bits) per 4x4 pixel block. A 4x4 block of GL_LUMINANCE8 data requires 16 bytes (1 byte per texel). This is a 2-to-1 compression ratio.

The LATC2 formats are 16 bytes (128 bits) per 4x4 pixel block. A 4x4 block of GL_LUMINANCE8_ALPHA8 data requires 32 bytes (2 bytes per texel). This is again a 2-to-1 compression ratio.

In contrast, the comparable compression ratio for the S3TC formats is 4-to-1.

Arguably, the lower compression ratio allows better compression quality particularly because the LATC formats compress each component separately.

10) *How do these new formats compare with the existing GL_LUMINANCE4, GL_LUMINANCE4_ALPHA4, and GL_LUMINANCE6_ALPHA2 internal formats?*

RESOLVED: The existing GL_LUMINANCE4, GL_LUMINANCE4_ALPHA4, and GL_LUMINANCE6_ALPHA2 internal formats provide a similar 2-to-1 compression ratio but mandate a uniform quantization for all components. In contrast, this extension provides a compression format with 3-bit quantization over a specifiable min/max range that can vary per 4x4 texel tile.

Additionally, many OpenGL implementations do not natively support the GL_LUMINANCE4, GL_LUMINANCE4_ALPHA4, and GL_LUMINANCE6_ALPHA2 internal formats but rather silently promote these formats to store 8 bits per component, thereby eliminating any storage/bandwidth advantage for these formats.

    11) *Does this extension require EXT_texture_compression_s3tc?*

        RESOLVED: No.

        As written, this specification does not rely on wording of the
        EXT_texture_compression_s3tc extension. For example, certain
        discussion added to Sections 3.8.2 and 3.8.3 is quite similar
        to corresponding EXT_texture_compression_s3tc language.

    12) *Should anything be said about the precision of texture filtering
        for these new formats?*

        RESOLVED: No precision requirements are part of the specification
        language since OpenGL extensions typically leave precision
        details to the implementation.

        Realistically, at least 8-bit filtering precision can be expected
        from implementations (and probably more).

    13) *Should these formats be allowed to specify 3D texture images
        when NV_texture_compression_vtc is supported?*

        RESOLVED: The NV_texture_compression_vtc stacks 4x4 blocks into
        4x4x4 bricks. It may be more desirable to represent compressed
        3D textures as simply slices of 4x4 blocks.

        However the NV_texture_compression_vtc extension expects
        data passed to the glCompressedTexImage commands to be "bricked"
        rather than blocked slices.

    14) *Why is GL_NV_texture_compression_latc also listed in the Name Strings
        section?*

        The very first GeForce 8800 driver shipped with the extension
        designated as NV before EXT-ization with S3 was agreed.
        Subsequent NVIDIA drivers will rename the extension to its EXT
        name only.

    15) *Should the the generic formats
        GL_COMPRESSED_LUMINANCE and GL_COMPRESSED_LUMINANCE_ALPHA
        correspond to COMPRESSED_LUMINANCE_LATC1_EXT and
        COMPRESSED_LUMINANCE_ALPHA_LATC2_EXT respecitively when this
        extension is supported?*

        RESOLVED: Yes. While no generic compression is strictly
        required for an implementation and there might exist superior
        compression schemes for luminance and luminance-alpha textures
        in the future, an application should reasonably expect that an
        implementation that supports EXT_texture_compression_latc will
        also use these formats for the generic compressed luminance and
        luminance-alpha formats.

        The COMPRESSED_LUMINANCE_LATC1_EXT and
        COMPRESSED_LUMINANCE_ALPHA_LATC2_EXT are generic enough in their
        respective luminance and luminance-alpha behavior that these
        compression formats are acceptable generic compressed formats
        for luminance and luminance-alpha generic compressed formats.

16) Should the GL_NUM_COMPRESSED_TEXTURE_FORMATS and GL_COMPRESSED_TEXTURE_FORMATS queries return the LATC formats?

RESOLVED: No.

The OpenGL 2.1 specification says "The only values returned by this query [GL_COMPRESSED_TEXTURE_FORMATS"] are those corresponding to formats suitable for general-purpose usage. The renderer will not enumerate formats with restrictions that need to be specifically understood prior to use."

Historically, OpenGL implementation have advertised the RGB and RGBA versions of the S3TC extensions compressed format tokens through this mechanism.

The specification is not sufficiently clear about what "suitable for general-purpose usage" means. Historically that seems to mean unsigned RGB or unsigned RGBA. The DXT1 format supporting alpha (GL_COMPRESSED_RGBA_S3TC_DXT1_EXT) is not exposed in the list (at least for NVIDIA drivers) because the alpha is always 1.0 expect when it is 0.0 when RGB is required to be black. NVIDIA's even limits itself to true linear RGB or RGBA formats, specifically not including EXT_texture_sRGB's sRGB S3TC compressed formats.

Adding luminance and luminance-alpha texture formats (and certainly signed versions of luminance and luminance-alpha formats!) invites potential comptaibility problems with old applications using this mechanism since old applications are unlikely to expect non-RGB or non-RGBA formats to be advertised through this mechanism. However no specific misinteractions with old applications is known.

Applications that seek to use the LATC formats should do so by looking for this extension's name in the string returned by glGetString(GL_EXTENSIONS) rather than what GL_NUM_COMPRESSED_TEXTURE_FORMATS and GL_COMPRESSED_TEXTURE_FORMATS return.

Revision History

Revision 1.1, April 24, 2007: mjk
- Add caveat about how signed LA decompression happens when lum0 equals -127 and lum1 equals -128. This caveat matches a decoding allowance in DirectX 10.

Revision 1.2, January 21, 2008: mjk
- Add issues #15 and #16.

Name

    EXT_texture_compression_rgtc

Name Strings

    GL_EXT_texture_compression_rgtc

Contributors

Mark J. Kilgard, NVIDIA
Pat Brown, NVIDIA
Yanjun Zhang, S3
Attila Barsi, Holografika

Contact

Mark J. Kilgard, NVIDIA Corporation (mjk 'at' nvidia.com)

Status

    Shipping for GeForce 8 Series (November 2006, Release 95)

Version

Date: January 21, 2008
Revision: 1.2

Number

    332

Dependencies

    OpenGL 1.3 or ARB_texture_compression required

    This extension is written against the OpenGL 2.0 (September 7,
    2004) specification.

Overview

    This extension introduces four new block-based texture compression
    formats suited for unsigned and signed red and red-green textures
    (hence the name "rgtc" for Red-Green Texture Compression).

    These formats are designed to reduce the storage requirements
    and memory bandwidth required for red and red-green textures by
    a factor of 2-to-1 over conventional uncompressed luminance and
    luminance-alpha textures with 8-bit components (GL_LUMINANCE8 and
    GL_LUMINANCE8_ALPHA8).

    The compressed signed red-green format is reasonably suited for
    storing compressed normal maps.

    This extension uses the same compression format as the
    EXT_texture_compression_latc extension except the color data is stored
    in the red and green components rather than luminance and alpha.

Representing compressed red and green components is consistent with the BC4 and BC5 compressed formats supported by DirectX 10.

New Procedures and Functions

None.

New Tokens

Accepted by the <internalformat> parameter of TexImage2D, CopyTexImage2D, and CompressedTexImage2D and the <format> parameter of CompressedTexSubImage2D:

```
    COMPRESSED_RED_RGTC1_EXT                        0x8DBB
    COMPRESSED_SIGNED_RED_RGTC1_EXT                 0x8DBC
    COMPRESSED_RED_GREEN_RGTC2_EXT                  0x8DBD
    COMPRESSED_SIGNED_RED_GREEN_RGTC2_EXT           0x8DBE
```

Additions to Chapter 2 of the OpenGL 2.0 Specification (OpenGL Operation)

None.

Additions to Chapter 3 of the OpenGL 2.0 Specification (Rasterization)

-- Section 3.8.1, Texture Image Specification

Add to Table 3.17 (page 155): Specific compressed internal formats

Compressed Internal Format	Base Internal Format
COMPRESSED_RED_RGTC1_EXT	RGB
COMPRESSED_SIGNED_RED_RGTC1_EXT	RGB
COMPRESSED_RED_GREEN_RGTC2_EXT	RGB
COMPRESSED_SIGNED_RED_GREEN_RGTC2_EXT	RGB

-- Section 3.8.2, Alternative Texture Image Specification Commands

Add to the end of the section (page 163):

"If the internal format of the texture image being modified is COMPRESSED_RED_RGTC1_EXT, COMPRESSED_SIGNED_RED_RGTC1_EXT, COMPRESSED_RED_GREEN_RGTC2_EXT, or COMPRESSED_SIGNED_RED_GREEN_RGTC2_EXT, the texture is stored using one of the two RGTC compressed texture image encodings (see appendix). Such images are easily edited along 4x4 texel boundaries, so the limitations on TexSubImage2D or CopyTexSubImage2D parameters are relaxed. TexSubImage2D and CopyTexSubImage2D will result in an INVALID_OPERATION error only if one of the following conditions occurs:

* <width> is not a multiple of four or equal to TEXTURE_WIDTH, unless <xoffset> and <yoffset> are both zero.
* <height> is not a multiple of four or equal to TEXTURE_HEIGHT, unless <xoffset> and <yoffset> are both zero.
* <xoffset> or <yoffset> is not a multiple of four.

The contents of any 4x4 block of texels of an RGTC compressed texture image that does not intersect the area being modified are preserved during valid TexSubImage2D and CopyTexSubImage2D calls."

-- Section 3.8.3, Compressed Texture Images

Add after the 4th paragraph (page 164) at the end of the CompressedTexImage discussion:

"If <internalformat> is COMPRESSED_RED_RGTC1_EXT, COMPRESSED_SIGNED_RED_RGTC1_EXT, COMPRESSED_RED_GREEN_RGTC2_EXT, or COMPRESSED_SIGNED_RED_GREEN_RGTC2_EXT, the compressed texture is stored using one of several RGTC compressed texture image formats. The RGTC texture compression algorithm supports only 2D images without borders. CompressedTexImage1D and CompressedTexImage3D produce an INVALID_ENUM error if <internalformat> is an RGTC format. CompressedTexImage2D will produce an INVALID_OPERATION error if <border> is non-zero.

Add to the end of the section (page 166) at the end of the CompressedTexSubImage discussion:

"If the internal format of the texture image being modified is COMPRESSED_RED_RGTC1_EXT, COMPRESSED_SIGNED_RED_RGTC1_EXT, COMPRESSED_RED_GREEN_RGTC2_EXT, or COMPRESSED_SIGNED_RED_GREEN_RGTC2_EXT, the texture is stored using one of the several RGTC compressed texture image formats. Since the RGTC texture compression algorithm supports only 2D images, CompressedTexSubImage1D and CompressedTexSubImage3D produce an INVALID_ENUM error if <format> is an RGTC format. Since RGTC images are easily edited along 4x4 texel boundaries, the limitations on CompressedTexSubImage2D are relaxed. CompressedTexSubImage2D will result in an INVALID_OPERATION error only if one of the following conditions occurs:

* <width> is not a multiple of four or equal to TEXTURE_WIDTH.
* <height> is not a multiple of four or equal to TEXTURE_HEIGHT.
* <xoffset> or <yoffset> is not a multiple of four.

The contents of any 4x4 block of texels of an RGTC compressed texture image that does not intersect the area being modified are preserved during valid TexSubImage2D and CopyTexSubImage2D calls."

-- Section 3.8.8, Texture Minification

Add a sentence to the last paragraph (page 174) just prior to the "Mipmapping" subheading:

"If the texture's internal format lacks components that exist in the texture's base internal format, such components are considered zero when the texture border color is sampled. (So despite the RGB base internal format of the COMPRESSED_RED_RGTC1_EXT and COMPRESSED_SIGNED_RED_RGTC1_EXT formats, the green and blue components of the texture border color are always considered zero. Likewise for the COMPRESSED_RED_GREEN_RGTC2_EXT, and COMPRESSED_SIGNED_RED_GREEN_RGTC2_EXT formats, the blue component is always considered zero.)"

Additions to Chapter 4 of the OpenGL 2.0 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 2.0 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 2.0 Specification (State and State Requests)

None.

Additions to Appendix A of the OpenGL 2.0 Specification (Invariance)

None.

Additions to the AGL/GLX/WGL Specifications

None.

GLX Protocol

None.

Dependencies on ARB_texture_compression

If ARB_texture_compression is supported, all the errors and accepted tokens for CompressedTexImage1D, CompressedTexImage2D, CompressedTexImage3D, CompressedTexSubImage1D, CompressedTexSubImage2D, and CompressedTexSubImage3D also apply respectively to the ARB-suffixed CompressedTexImage1DARB, CompressedTexImage2DARB, CompressedTexImage3DARB, CompressedTexSubImage1DARB, CompressedTexSubImage2DARB, and CompressedTexSubImage3DARB.

Errors

INVALID_ENUM is generated by CompressedTexImage1D or CompressedTexImage3D if <internalformat> is COMPRESSED_LUMINANCE_LACT1_EXT, COMPRESSED_SIGNED_RED_RGTC1_EXT, COMPRESSED_RED_GREEN_RGTC2_EXT, or COMPRESSED_SIGNED_RED_GREEN_RGTC2_EXT.

INVALID_OPERATION is generated by CompressedTexImage2D if <internalformat> is COMPRESSED_LUMINANCE_LACT1_EXT, COMPRESSED_SIGNED_RED_RGTC1_EXT, COMPRESSED_RED_GREEN_RGTC2_EXT, or COMPRESSED_SIGNED_RED_GREEN_RGTC2_EXT and <border> is not equal to zero.

INVALID_ENUM is generated by CompressedTexSubImage1D or CompressedTexSubImage3D if <format> is COMPRESSED_LUMINANCE_LACT1_EXT, COMPRESSED_SIGNED_RED_RGTC1_EXT, COMPRESSED_RED_GREEN_RGTC2_EXT, or COMPRESSED_SIGNED_RED_GREEN_RGTC2_EXT.

```
    INVALID_OPERATION is generated by TexSubImage2D
    CopyTexSubImage2D, or CompressedTexSubImage2D if
    TEXTURE_INTERNAL_FORMAT is COMPRESSED_LUMINANCE_LACT1_EXT,
    COMPRESSED_SIGNED_RED_RGTC1_EXT, COMPRESSED_RED_GREEN_RGTC2_EXT,
    or COMPRESSED_SIGNED_RED_GREEN_RGTC2_EXT and any of the following
    apply: <width> is not a multiple of four or equal to TEXTURE_WIDTH;
    <height> is not a multiple of four or equal to TEXTURE_HEIGHT;
    <xoffset> or <yoffset> is not a multiple of four.


    The following restrictions from the ARB_texture_compression
    specification do not apply to RGTC texture formats, since subimage
    modification is straightforward as long as the subimage is properly
    aligned.

    DELETE: INVALID_OPERATION is generated by TexSubImage1D, TexSubImage2D,
    DELETE: TexSubImage3D, CopyTexSubImage1D, CopyTexSubImage2D, or
    DELETE: CopyTexSubImage3D if the internal format of the texture image is
    DELETE: compressed and <xoffset>, <yoffset>, or <zoffset> does not equal
    DELETE: -b, where b is value of TEXTURE_BORDER.

    DELETE: INVALID_VALUE is generated by CompressedTexSubImage1D,
    DELETE: CompressedTexSubImage2D, or CompressedTexSubImage3D if the
    DELETE: entire texture image is not being edited:  if <xoffset>,
    DELETE: <yoffset>, or <zoffset> is greater than -b, <xoffset> + <width> is
    DELETE: less than w+b, <yoffset> + <height> is less than h+b, or <zoffset>
    DELETE: + <depth> is less than d+b, where b is the value of
    DELETE: TEXTURE_BORDER, w is the value of TEXTURE_WIDTH, h is the value of
    DELETE: TEXTURE_HEIGHT, and d is the value of TEXTURE_DEPTH.

    See also errors in the GL_ARB_texture_compression specification.
```

New State

```
    4 new state values are added for the per-texture object
    GL_TEXTURE_INTERNAL_FORMAT state.

    In the "Textures" state table( page 278), increment the
    TEXTURE_INTERNAL_FORMAT subscript for Z by 4 in the "Type" row.

    [NOTE: The OpenGL 2.0 specification actually should read "n x Z48*"
    because of the 6 generic compressed internal formats in table 3.18.]
```

New Implementation Dependent State

```
    None
```

Appendix

RGTC Compressed Texture Image Formats

```
    Compressed texture images stored using the RGTC compressed image
    encodings are represented as a collection of 4x4 texel blocks,
    where each block contains 64 or 128 bits of texel data.  The image
    is encoded as a normal 2D raster image in which each 4x4 block is
    treated as a single pixel.  If an RGTC image has a width or height
    less than four, the data corresponding to texels outside the image
    are irrelevant and undefined.

    When an RGTC image with a width of <w>, height of <h>, and block
    size of <blocksize> (8 or 16 bytes) is decoded, the corresponding
    image size (in bytes) is:

        ceil(<w>/4) * ceil(<h>/4) * blocksize.

    When decoding an RGTC image, the block containing the texel at offset
    (<x>, <y>) begins at an offset (in bytes) relative to the base of the
    image of:

        blocksize * (ceil(<w>/4) * floor(<y>/4) + floor(<x>/4)).

    The data corresponding to a specific texel (<x>, <y>) are extracted
    from a 4x4 texel block using a relative (x,y) value of

        (<x> modulo 4, <y> modulo 4).

    There are four distinct RGTC image formats:


    COMPRESSED_RED_RGTC1:  Each 4x4 block of texels consists of
    64 bits of unsigned red image data.

    Each red image data block is encoded as a sequence of 8 bytes, called
    (in order of increasing address):

            red0, red1, bits_0, bits_1, bits_2, bits_3, bits_4, bits_5

        The 6 "bits_*" bytes of the block are decoded into a 48-bit bit
        vector:

            bits   = bits_0 +
                     256 * (bits_1 +
                            256 * (bits_2 +
                                   256 * (bits_3 +
                                          256 * (bits_4 +
                                                 256 * bits_5))))

        red0 and red1 are 8-bit unsigned integers that are unpacked to red
        values RED0 and RED1 as though they were pixels with a <format>
        of LUMINANCE and a type of UNSIGNED_BTYE.

        bits is a 48-bit unsigned integer, from which a three-bit control
        code is extracted for a texel at location (x,y) in the block
        using:

            code(x,y) = bits[3*(4*y+x)+2..3*(4*y+x)+0]

        where bit 47 is the most significant and bit 0 is the least
        significant bit.

        The red value R for a texel at location (x,y) in the block is
        given by:
```

```
        RED0,              if red0 > red1 and code(x,y) == 0
        RED1,              if red0 > red1 and code(x,y) == 1
        (6*RED0+  RED1)/7, if red0 > red1 and code(x,y) == 2
        (5*RED0+2*RED1)/7, if red0 > red1 and code(x,y) == 3
        (4*RED0+3*RED1)/7, if red0 > red1 and code(x,y) == 4
        (3*RED0+4*RED1)/7, if red0 > red1 and code(x,y) == 5
        (2*RED0+5*RED1)/7, if red0 > red1 and code(x,y) == 6
        (  RED0+6*RED1)/7, if red0 > red1 and code(x,y) == 7

        RED0,              if red0 <= red1 and code(x,y) == 0
        RED1,              if red0 <= red1 and code(x,y) == 1
        (4*RED0+  RED1)/5, if red0 <= red1 and code(x,y) == 2
        (3*RED0+2*RED1)/5, if red0 <= red1 and code(x,y) == 3
        (2*RED0+3*RED1)/5, if red0 <= red1 and code(x,y) == 4
        (  RED0+4*RED1)/5, if red0 <= red1 and code(x,y) == 5
        MINRED,            if red0 <= red1 and code(x,y) == 6
        MAXRED,            if red0 <= red1 and code(x,y) == 7

    MINRED and MAXRED are 0.0 and 1.0 respectively.
```

Since the decoded texel has a red format, the resulting RGBA value
for the texel is (R,0,0,1).

COMPRESSED_SIGNED_RED_RGTC1: Each 4x4 block of texels consists of
64 bits of signed red image data. The red values of a texel are
extracted in the same way as COMPRESSED_RED_RGTC1 except red0, red1,
RED0, RED1, MINRED, and MAXRED are signed values defined as follows:

```
    red0 and red1 are 8-bit signed (two's complement) integers.

           { red0 / 127.0, red0 > -128
    RED0 = {
           { -1.0,         red0 == -128

           { red1 / 127.0, red1 > -128
    RED1 = {
           { -1.0,         red1 == -128

    MINRED = -1.0

    MAXRED =  1.0
```

CAVEAT for signed red0 and red1 values: the expressions "red0 >
red1" and "red0 <= red1" above are considered undefined (read: may
vary by implementation) when red0 equals -127 and red1 equals -128,
This is because if red0 were remapped to -127 prior to the comparison
to reduce the latency of a hardware decompressor, the expressions
would reverse their logic. Encoders for the signed LA formats should
avoid encoding blocks where red0 equals -127 and red1 equals -128.

COMPRESSED_RED_GREEN_RGTC2: Each 4x4 block of texels consists of
64 bits of compressed unsigned red image data followed by 64 bits
of compressed unsigned green image data.

The first 64 bits of compressed red are decoded exactly like COMPRESSED_RED_RGTC1 above.

The second 64 bits of compressed green are decoded exactly like COMPRESSED_RED_RGTC1 above except the decoded value R for this second block is considered the resulting green value G.

Since the decoded texel has a red-green format, the resulting RGBA value for the texel is (R,G,0,1).

COMPRESSED_SIGNED_RED_GREEN_RGTC2: Each 4x4 block of texels consists of 64 bits of compressed signed red image data followed by 64 bits of compressed signed green image data.

The first 64 bits of compressed red are decoded exactly like COMPRESSED_SIGNED_RED_RGTC1 above.

The second 64 bits of compressed green are decoded exactly like COMPRESSED_SIGNED_RED_RGTC1 above except the decoded value R for this second block is considered the resulting green value G.

Since this image has a red-green format, the resulting RGBA value is (R,G,0,1).

Issues

1) *What should these new formats be called?*

   RESOLVED: "rgtc" for Red-Green Texture Compression.

2) *How should the uncompressed and filtered texels be returned by texture fetches?*

   RESOLVED: Red values show up as (R,0,0,1) where R is the red value, green and blue are forced to 0, and alpha is forced to 1. Likewise, red-green values show up as (R,G,0,1) where G is the green value.

   Prior extensions such as NV_float_buffer and NV_texture_shader have introduced formats such as GL_FLOAT_R_NV and GL_DSDT_NV where one- and two-component texture formats show up as (X,0,0,1) or (X,Y,0,1) RGBA texels. The RGTC formats mimic these two-component formats.

   The (X,Y,0,1) convention, particularly with signed components, is nice for normal maps because a normalized vector can be formed by a shader program by computing sqrt(abs(1-X*X-Y*Y)) for the Z component.

   While GL_RED is a valid external format, core OpenGL provides no GL_RED_GREEN external format. Applications can either use GL_RGB or GL_RGBA and pad out the blue and alpha components, or use the two-component GL_LUMINANCE_ALPHA color format and use the color matrix functionality to swizzle the luminance and alpha values into red and green respectively.

3) *Should red and red-green compression formats with signed components be introduced when the core specification lacked uncompressed red and red-green texture formats?*

RESOLVED: Yes, signed red and red-green compression formats should be added.

Signed red-green formats are suited for compressed normal maps. Compressed normal maps may well be the dominant use of this extension.

Unsigned red-green formats require an extra "expand normal" operation to convert [0,1] to [-1,+1]. Direct support for signed red-green formats avoids this step in a shader program.

4) *Should there be a mix of signed red and unsigned green or vice versa?*

RESOLVED: No.

NV_texture_shader provided an internal format (GL_SIGNED_RGB_UNSIGNED_ALPHA_NV) with mixed signed and unsigned components. The format saw little usage. There's no reason to think a GL_SIGNED_RED_UNSIGNED_GREEN format would be any more useful or popular.

5) *How are signed integer values mapped to floating-point values?*

RESOLVED: A signed 8-bit two's complement value X is computed to a floating-point value Xf with the formula:

```
         { X / 127.0, X > -128
    Xf = {
         { -1.0,      X == -128
```

This conversion means -1, 0, and +1 are all exactly representable, however -128 and -127 both map to -1.0. Mapping -128 to -1.0 avoids the numerical awkwardness of have a representable value slightly more negative than -1.0.

This conversion is intentionally NOT the "byte" conversion listed in Table 2.9 for component conversions. That conversion says:

```
    Xf = (2*X + 1) / 255.0
```

The Table 2.9 conversion is incapable of exactly representing zero.

6) *How will signed components resulting from GL_COMPRESSED_SIGNED_RED_RGTC1_EXT and GL_COMPRESSED_SIGNED_RED_GREEN_RGTC2_EXT texture fetches interact with fragment coloring?*

RESOLVED: The specification language for this extension is silent about clamping behavior leaving this to the core specification and other extensions. The clamping or lack of clamping is left to the core specification and other extensions.

For assembly program extensions supporting texture fetches (ARB_fragment_program, NV_fragment_program, NV_vertex_program3, etc.) or the OpenGL Shading Language, these signed formats will appear as expected with unclamped signed components as a result of a texture fetch instruction.

If ARB_color_buffer_float is supported, its clamping controls will apply.

NV_texture_shader extension, if supported, adds support for fixed-point textures with signed components and relaxed the fixed-function texture environment clamping appropriately. If the NV_texture_shader extension is supported, its specified behavior for the texture environment applies where intermediate values are clamped to [-1,1] unless stated otherwise as in the case of explicitly clamped to [0,1] for GL_COMBINE. or clamping the linear interpolation weight to [0,1] for GL_DECAL and GL_BLEND.

Otherwise, the conventional core texture environment clamps incoming, intermediate, and output color components to [0,1].

This implies that the conventional texture environment functionality of unextended OpenGL 1.5 or OpenGL 2.0 without using GLSL (and with none of the extensions referred to above) is unable to make proper use of the signed texture formats added by this extension because the conventional texture environment requires texture source colors to be clamped to [0,1]. Texture filtering of these signed formats would be still signed, but negative values generated post-filtering would be clamped to zero by the core texture environment functionality. The expectation is clearly that this extension would be co-implemented with one of the previously referred to extensions or used with GLSL for the new signed formats to be useful.

7) *Should a specific normal map compression format be added?*

RESOLVED: No.

It's probably short-sighted to design a format just for normal maps. Indeed, NV_texture_shader added a GL_SIGNED_HILO_NV format with exactly the kind of "hemisphere remap" useful for normal maps and the format went basically unused. Instead, this extension provides the mechanism for compressed normal maps based on the more conventional red-green format.

The GL_COMPRESSED_RED_GREEN_RGTC2_EXT and GL_COMPRESSED_SIGNED_RED_GREEN_RGTC2_EXT formats are sufficient for normal maps with additional shader instructions used to generate the 3rd component.

8) *Should uncompressed signed red and red-green formats be added by this extension?*

RESOLVED: No, this extension is focused on just adding compressed texture formats.

The NV_texture_shader extension adds such uncompressed signed texture formats. A distinct multi-vendor extension for signed fixed-point texture formats could provide all or a subset of the signed fixed-point uncompressed texture formats introduced by NV_texture_shader.

9) *What compression ratios does this extension provide?*

The RGTC1 formats are 8 bytes (64 bits) per 4x4 pixel block. A 4x4 block of GL_LUMINANCE8 data requires 16 bytes (1 byte per texel). This is a 2-to-1 compression ratio.

The RGTC2 formats are 16 bytes (128 bits) per 4x4 pixel block. A 4x4 block of GL_LUMINANCE8_ALPHA8 data requires 32 bytes (2 bytes per texel). This is again a 2-to-1 compression ratio.

In contrast, the comparable compression ratio for the S3TC formats is 4-to-1.

Arguably, the lower compression ratio allows better compression quality particularly because the RGTC formats compress each component separately.

10) *How do these new formats compare with the existing GL_LUMINANCE4, GL_LUMINANCE4_ALPHA4, and GL_LUMINANCE6_ALPHA2 internal formats?*

RESOLVED: The existing GL_LUMINANCE4, GL_LUMINANCE4_ALPHA4, and GL_LUMINANCE6_ALPHA2 internal formats provide a similar 2-to-1 compression ratio but mandate a uniform quantization for all components. In contrast, this extension provides a compression format with 3-bit quantization over a specifiable min/max range that can vary per 4x4 texel tile.

Additionally, many OpenGL implementations do not natively support the GL_LUMINANCE4, GL_LUMINANCE4_ALPHA4, and GL_LUMINANCE6_ALPHA2 internal formats but rather silently promote these formats to store 8 bits per component, thereby eliminating any storage/bandwidth advantage for these formats.

11) *Does this extension require EXT_texture_compression_s3tc?*

RESOLVED: No.

As written, this specification does not rely on wording of the EXT_texture_compression_s3tc extension. For example, certain discussion added to Sections 3.8.2 and 3.8.3 is quite similar to corresponding EXT_texture_compression_s3tc language.

12) *Should anything be said about the precision of texture filtering for these new formats?*

RESOLVED: No precision requirements are part of the specification language since OpenGL extensions typically leave precision details to the implementation.

Realistically, at least 8-bit filtering precision can be expected from implementations (and probably more).

13) *Should these formats be allowed to specify 3D texture images when NV_texture_compression_vtc is supported?*

RESOLVED: The NV_texture_compression_vtc stacks 4x4 blocks into 4x4x4 bricks. It may be more desirable to represent compressed 3D textures as simply slices of 4x4 blocks.

However the NV_texture_compression_vtc extension expects data passed to the glCompressedTexImage commands to be "bricked" rather than blocked slices.

14) *How is the texture border color handled for the blue component of an RGTC2 texture and the green and blue components of an RGTC1 texture?*

RESOLVED: The base texture format is RGB for the RGTC1 and RGTC2 texture formats. This would mean table 3.15 would be used to determine how the texture border color is interpreted and which components are considered.

However since only red or red/green components exist for the RGTC1 and RGTC2 formats, it makes little sense to require the blue component be supplied by the texture border color and hence be involved (meaningfully only when the border is sampled) in texture filtering.

For this reason, a statement is added to section 3.8.8 says that if a texture's internal format lacks components that exist in the texture's base internal format, such components contain zero (ignoring the texture's corresponding texture border color component value) when the texture border color is sampled.

So the green and blue components of the filtered result of a RGTC1 texture are always zero, even when the border is sampled. Similarly the blue component of the filtered result of a RGTC2 texture is always zero, even when the border is sampled.

15) What should glGetTexLevelParameter return for *GL_TEXTURE_GREEN_SIZE and GL_TEXTURE_BLUE_SIZE for the RGTC1 formats? What should glGetTexLevelParameter return for GL_TEXTURE_BLUE_SIZE for the RGTC2 formats?*

RESOLVED: Zero bits.

These formats always return 0.0 for these respective components and have no bits devoted to these components.

Returning 8 bits for red size of RGTC1 and the red and green sizes of RGTC2 makes sense because that's the maximum potential precision for the uncompressed texels.

```
    16) Should the token names contain R and RG or RED and RED_GREEN?

        RESOLVED:  RED and RED_GREEN.

        Saying RGB and RGBA makes sense for three- and four-component
        formats rather than spelling out the component names because
        RGB and RGBA are used so commonly and spelling out the names it
        too wordy.

        But for 1- and 2-component names, we follow the precedent by
        GL_LUMINANCE and GL_LUMINANCE_ALPHA.  This extension spells out
        the component names of 1- and 2-component names.

        Another reason to avoid R and RG is the existing meaning of
        the GL_R and GL_RED tokens.  GL_RED already exists as a token
        name for a single-component external format.  GL_R also already
        exists as a token name but refers to the R texture coordinate,
        not the red color component.

    17) Can you use the GL_RED external format with glTexImage2D and other
        such commands to load textures with the
        GL_COMPRESSED_RED_RGTC1_EXT or GL_COMPRESSED_SIGNED_RED_RGTC1_EXT
        internal formats?

        RESOLVED: Yes.

        GL_RED has been a valid external format parameter to glTexImage
        and similar commands since OpenGL 1.0.

    18) Should any of the generic compression GL_COMPRESSED_* tokens in
        OpenGL 2.1 map to RGTC formats?

        RESOLVED:  No.  The RGTC formats are missing color components
        so are not adequate implementations for any of the generic
        compression formats.

    19) Should the GL_NUM_COMPRESSED_TEXTURE_FORMATS and
        GL_COMPRESSED_TEXTURE_FORMATS queries return the RGTC formats?

        RESOLVED:  No.

        The OpenGL 2.1 specification says "The only values returned
        by this query [GL_COMPRESSED_TEXTURE_FORMATS"] are those
        corresponding to formats suitable for general-purpose usage.
        The renderer will not enumerate formats with restrictions that
        need to be specifically understood prior to use."

        Compressed textures with just red or red-green components are
        not general-purpose so should not be returned by these queries
        because they have restrictions.

        Applications that seek to use the RGTC formats should do so
        by looking for this extension's name in the string returned by
        glGetString(GL_EXTENSIONS) rather than
        what GL_NUM_COMPRESSED_TEXTURE_FORMATS and
        GL_COMPRESSED_TEXTURE_FORMATS return.
```

Revision History

```
    Revision 1.1, April 24, 2007: mjk
        -  Add caveat about how signed LA decompression happens when
           lum0 equals -127 and lum1 equals -128.  This caveat matches
           a decoding allowance in DirectX 10.

    Revision 1.2, January 21, 2008: mjk
        -  Add issues #18 and #19.
```

Name

    EXT_texture_compression_s3tc

Name Strings

    GL_EXT_texture_compression_s3tc

Status

    FINAL

Version

    1.3, 07 June 2007 (containing only clarifications relative to
                       version 1.0, dated 7 July 2000)

Number

    198

Dependencies

    OpenGL 1.1 is required.

    GL_ARB_texture_compression is required.

    This extension is written against the OpenGL 1.2.1 Specification.

Overview

    This extension provides additional texture compression functionality
    specific to S3's S3TC format (called DXTC in Microsoft's DirectX API),
    subject to all the requirements and limitations described by the extension
    GL_ARB_texture_compression.

    This extension supports DXT1, DXT3, and DXT5 texture compression formats.
    For the DXT1 image format, this specification supports an RGB-only mode
    and a special RGBA mode with single-bit "transparent" alpha.

IP Status

    Contact S3 Incorporated (http://www.s3.com) regarding any intellectual
    property issues associated with implementing this extension.

    WARNING: Vendors able to support S3TC texture compression in Direct3D
    drivers do not necessarily have the right to use the same functionality in
    OpenGL.

Issues

    (1) Should DXT2 and DXT4 (premultiplied alpha) formats be supported?

        RESOLVED: No -- insufficient interest. Supporting DXT2 and DXT4
        would require some rework to the TexEnv definition (maybe add a new
        base internal format RGBA_PREMULTIPLIED_ALPHA) for these formats.
        Note that the EXT_texture_env_combine extension (which extends normal
        TexEnv modes) can be used to support textures with premultipled alpha.

    *(2) Should generic "RGB_S3TC_EXT" and "RGBA_S3TC_EXT" enums be supported
        or should we use only the DXT<n> enums?*

        RESOLVED: No. A generic RGBA_S3TC_EXT is problematic because DXT3 and DXT5 are both nominally RGBA (and DXT1 with the 1-bit alpha is also) yet one format must be chosen up front.

(3) *Should TexSubImage support all block-aligned edits or just the minimal functionality required by the ARB_texture_compression extension?*

RESOLVED: Allow all valid block-aligned edits.

(4) *A pre-compressed image with a DXT1 format can be used as either an RGB_S3TC_DXT1 or an RGBA_S3TC_DXT1 image. If the image has transparent texels, how are they treated in each format?*

RESOLVED: The renderer has to make sure that an RGB_S3TC_DXT1 format is decoded as RGB (where alpha is effectively one for all texels), while RGBA_S3TC_DXT1 is decoded as RGBA (where alpha is zero for all texels with "transparent" encodings). Otherwise, the formats are identical.

(5) *Is the encoding of the RGB components for DXT1 formats correct in this spec? MSDN documentation does not specify an RGB color for the "transparent" encoding. Is it really black?*

RESOLVED: Yes. The specification for the DXT1 format initially required black, but later changed that requirement to a recommendation. All vendors involved in the definition of this specification support black. In addition, specifying black has a useful behavior.

When blending multiple texels (GL_LINEAR filtering), mixing opaque and transparent samples is problematic. Defining a black color on transparent texels achieves a sensible result that works like a texture with premultiplied alpha. For example, if three opaque white and one transparent sample is being averaged, the result would be a 75% intensity gray (with an alpha of 75%). This is the same result on the color channels as would be obtained using a white color, 75% alpha, and a SRC_ALPHA blend factor.

(6) *Is the encoding of the RGB components for DXT3 and DXT5 formats correct in this spec? MSDN documentation suggests that the RGB blocks for DXT3 and DXT5 are decoded as described by the DXT1 format.*

RESOLVED: Yes -- this appears to be a bug in the MSDN documentation. The specification for the DXT2-DXT5 formats require decoding using the opaque block encoding, regardless of the relative values of "color0" and "color1".

New Procedures and Functions

None.

New Tokens

Accepted by the <internalformat> parameter of TexImage2D, CopyTexImage2D, and CompressedTexImage2DARB and the <format> parameter of CompressedTexSubImage2DARB:

```
    COMPRESSED_RGB_S3TC_DXT1_EXT                  0x83F0
    COMPRESSED_RGBA_S3TC_DXT1_EXT                 0x83F1
    COMPRESSED_RGBA_S3TC_DXT3_EXT                 0x83F2
    COMPRESSED_RGBA_S3TC_DXT5_EXT                 0x83F3
```

Additions to Chapter 2 of the OpenGL 1.2.1 Specification (OpenGL Operation)

None.

Additions to Chapter 3 of the OpenGL 1.2.1 Specification (Rasterization)

Add to Table 3.16.1: Specific Compressed Internal Formats

Compressed Internal Format	Base Internal Format
COMPRESSED_RGB_S3TC_DXT1_EXT	RGB
COMPRESSED_RGBA_S3TC_DXT1_EXT	RGBA
COMPRESSED_RGBA_S3TC_DXT3_EXT	RGBA
COMPRESSED_RGBA_S3TC_DXT5_EXT	RGBA

Modify **Section 3.8.2, Alternate Image Specification**

(add to end of TexSubImage discussion, p.123 -- after edit from the ARB_texture_compression spec)

If the internal format of the texture image being modified is COMPRESSED_RGB_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT3_EXT, or COMPRESSED_RGBA_S3TC_DXT5_EXT, the texture is stored using one of the several S3TC compressed texture image formats. Such images are easily edited along 4x4 texel boundaries, so the limitations on TexSubImage2D or CopyTexSubImage2D parameters are relaxed. TexSubImage2D and CopyTexSubImage2D will result in an INVALID_OPERATION error only if one of the following conditions occurs:

* <width> is not a multiple of four or equal to TEXTURE_WIDTH, unless <xoffset> and <yoffset> are both zero.
* <height> is not a multiple of four or equal to TEXTURE_HEIGHT, unless <xoffset> and <yoffset> are both zero.
* <xoffset> or <yoffset> is not a multiple of four.

The contents of any 4x4 block of texels of an S3TC compressed texture image that does not intersect the area being modified are preserved during valid TexSubImage2D and CopyTexSubImage2D calls.

Add to Section 3.8.2, Alternate Image Specification (adding to the end of the CompressedTexImage section introduced by the ARB_texture_compression spec)

If <internalformat> is COMPRESSED_RGB_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT3_EXT, or COMPRESSED_RGBA_S3TC_DXT5_EXT, the compressed texture is stored using one of several S3TC compressed texture image formats. The S3TC texture compression algorithm supports only 2D images without borders. CompressedTexImage1DARB and CompressedTexImage3DARB produce an INVALID_ENUM error if <internalformat> is an S3TC format. CompressedTexImage2DARB will produce an INVALID_OPERATION error if <border> is non-zero.

Add to Section 3.8.2, Alternate Image Specification (adding to the end of the CompressedTexSubImage section introduced by the ARB_texture_compression spec)

If the internal format of the texture image being modified is COMPRESSED_RGB_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT3_EXT, or COMPRESSED_RGBA_S3TC_DXT5_EXT, the texture is stored using one of the several S3TC compressed texture image formats. Since the S3TC texture compression algorithm supports only 2D images, CompressedTexSubImage1DARB and CompressedTexSubImage3DARB produce an INVALID_ENUM error if <format> is an S3TC format. Since S3TC images are easily edited along 4x4 texel boundaries, the limitations on CompressedTexSubImage2D are relaxed. CompressedTexSubImage2D will result in an INVALID_OPERATION error only if one of the following conditions occurs:

* <width> is not a multiple of four or equal to TEXTURE_WIDTH.
* <height> is not a multiple of four or equal to TEXTURE_HEIGHT.
* <xoffset> or <yoffset> is not a multiple of four.

The contents of any 4x4 block of texels of an S3TC compressed texture image that does not intersect the area being modified are preserved during valid TexSubImage2D and CopyTexSubImage2D calls.

Additions to Chapter 4 of the OpenGL 1.2.1 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 1.2.1 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 1.2.1 Specification (State and State Requests)

None.

Additions to Appendix A of the OpenGL 1.2.1 Specification (Invariance)

None.

Additions to the AGL/GLX/WGL Specifications

None.

GLX Protocol

None.

Errors

INVALID_ENUM is generated by CompressedTexImage1DARB or CompressedTexImage3DARB if <internalformat> is COMPRESSED_RGB_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT3_EXT, or COMPRESSED_RGBA_S3TC_DXT5_EXT.

INVALID_OPERATION is generated by CompressedTexImage2DARB if <internalformat> is COMPRESSED_RGB_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT3_EXT, or COMPRESSED_RGBA_S3TC_DXT5_EXT and <border> is not equal to zero.

INVALID_ENUM is generated by CompressedTexSubImage1DARB or CompressedTexSubImage3DARB if <format> is COMPRESSED_RGB_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT3_EXT, or COMPRESSED_RGBA_S3TC_DXT5_EXT.

INVALID_OPERATION is generated by TexSubImage2D CopyTexSubImage2D, or CompressedTexSubImage2D if TEXTURE_INTERNAL_FORMAT is COMPRESSED_RGB_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT3_EXT, or COMPRESSED_RGBA_S3TC_DXT5_EXT and any of the following apply: <width> is not a multiple of four or equal to TEXTURE_WIDTH; <height> is not a multiple of four or equal to TEXTURE_HEIGHT; <xoffset> or <yoffset> is not a multiple of four.

The following restrictions from the ARB_texture_compression specification do not apply to S3TC texture formats, since subimage modification is straightforward as long as the subimage is properly aligned.

```
DELETE: INVALID_OPERATION is generated by TexSubImage1D, TexSubImage2D,
DELETE: TexSubImage3D, CopyTexSubImage1D, CopyTexSubImage2D, or
DELETE: CopyTexSubImage3D if the internal format of the texture image is
DELETE: compressed and <xoffset>, <yoffset>, or <zoffset> does not equal
DELETE: -b, where b is value of TEXTURE_BORDER.

DELETE: INVALID_VALUE is generated by CompressedTexSubImage1DARB,
DELETE: CompressedTexSubImage2DARB, or CompressedTexSubImage3DARB if the
DELETE: entire texture image is not being edited:  if <xoffset>,
DELETE: <yoffset>, or <zoffset> is greater than -b, <xoffset> + <width> is
DELETE: less than w+b, <yoffset> + <height> is less than h+b, or <zoffset>
DELETE: + <depth> is less than d+b, where b is the value of
DELETE: TEXTURE_BORDER, w is the value of TEXTURE_WIDTH, h is the value of
DELETE: TEXTURE_HEIGHT, and d is the value of TEXTURE_DEPTH.
```

See also errors in the GL_ARB_texture_compression specification.

New State

In the "Textures" state table, increment the TEXTURE_INTERNAL_FORMAT subscript for Z by 4 in the "Type" row.

New Implementation Dependent State

None

Appendix

S3TC Compressed Texture Image Formats

Compressed texture images stored using the S3TC compressed image formats are represented as a collection of 4x4 texel blocks, where each block contains 64 or 128 bits of texel data. The image is encoded as a normal 2D raster image in which each 4x4 block is treated as a single pixel. If an S3TC image has a width or height less than four, the data corresponding to texels outside the image are irrelevant and undefined.

When an S3TC image with a width of <w>, height of <h>, and block size of <blocksize> (8 or 16 bytes) is decoded, the corresponding image size (in bytes) is:

```
ceil(<w>/4) * ceil(<h>/4) * blocksize.
```

When decoding an S3TC image, the block containing the texel at offset (<x>, <y>) begins at an offset (in bytes) relative to the base of the image of:

```
blocksize * (ceil(<w>/4) * floor(<y>/4) + floor(<x>/4)).
```

The data corresponding to a specific texel (<x>, <y>) are extracted from a 4x4 texel block using a relative (x,y) value of

```
(<x> modulo 4, <y> modulo 4).
```

```
    There are four distinct S3TC image formats:

    COMPRESSED_RGB_S3TC_DXT1_EXT:  Each 4x4 block of texels consists of 64
    bits of RGB image data.

    Each RGB image data block is encoded as a sequence of 8 bytes, called (in
    order of increasing address):

            c0_lo, c0_hi, c1_lo, c1_hi, bits_0, bits_1, bits_2, bits_3

        The 8 bytes of the block are decoded into three quantities:

            color0 = c0_lo + c0_hi * 256
            color1 = c1_lo + c1_hi * 256
            bits   = bits_0 + 256 * (bits_1 + 256 * (bits_2 + 256 * bits_3))

        color0 and color1 are 16-bit unsigned integers that are unpacked to
        RGB colors RGB0 and RGB1 as though they were 16-bit packed pixels with
        a <format> of RGB and a type of UNSIGNED_SHORT_5_6_5.

        bits is a 32-bit unsigned integer, from which a two-bit control code
        is extracted for a texel at location (x,y) in the block using:

            code(x,y) = bits[2*(4*y+x)+1..2*(4*y+x)+0]

        where bit 31 is the most significant and bit 0 is the least
        significant bit.

        The RGB color for a texel at location (x,y) in the block is given by:

            RGB0,              if color0 > color1 and code(x,y) == 0
            RGB1,              if color0 > color1 and code(x,y) == 1
            (2*RGB0+RGB1)/3,   if color0 > color1 and code(x,y) == 2
            (RGB0+2*RGB1)/3,   if color0 > color1 and code(x,y) == 3

            RGB0,              if color0 <= color1 and code(x,y) == 0
            RGB1,              if color0 <= color1 and code(x,y) == 1
            (RGB0+RGB1)/2,     if color0 <= color1 and code(x,y) == 2
            BLACK,             if color0 <= color1 and code(x,y) == 3

        Arithmetic operations are done per component, and BLACK refers to an
        RGB color where red, green, and blue are all zero.

    Since this image has an RGB format, there is no alpha component and the
    image is considered fully opaque.


    COMPRESSED_RGBA_S3TC_DXT1_EXT:  Each 4x4 block of texels consists of 64
    bits of RGB image data and minimal alpha information.  The RGB components
    of a texel are extracted in the same way as COMPRESSED_RGB_S3TC_DXT1_EXT.

        The alpha component for a texel at location (x,y) in the block is
        given by:

            0.0,               if color0 <= color1 and code(x,y) == 3
            1.0,               otherwise

        IMPORTANT:  When encoding an RGBA image into a format using 1-bit
        alpha, any texels with an alpha component less than 0.5 end up with an
        alpha of 0.0 and any texels with an alpha component greater than or
        equal to 0.5 end up with an alpha of 1.0.  When encoding an RGBA image
        into the COMPRESSED_RGBA_S3TC_DXT1_EXT format, the resulting red,
        green, and blue components of any texels with a final alpha of 0.0
        will automatically be zero (black).  If this behavior is not desired
        by an application, it should not use COMPRESSED_RGBA_S3TC_DXT1_EXT.
        This format will never be used when a generic compressed internal
        format (Table 3.16.2) is specified, although the nearly identical
        format COMPRESSED_RGB_S3TC_DXT1_EXT (above) may be.


    COMPRESSED_RGBA_S3TC_DXT3_EXT:  Each 4x4 block of texels consists of 64
    bits of uncompressed alpha image data followed by 64 bits of RGB image
    data.

    Each RGB image data block is encoded according to the
    COMPRESSED_RGB_S3TC_DXT1_EXT format, with the exception that the two code
    bits always use the non-transparent encodings.  In other words, they are
    treated as though color0 > color1, regardless of the actual values of
    color0 and color1.

    Each alpha image data block is encoded as a sequence of 8 bytes, called
    (in order of increasing address):

            a0, a1, a2, a3, a4, a5, a6, a7

        The 8 bytes of the block are decoded into one 64-bit integer:

            alpha = a0 + 256 * (a1 + 256 * (a2 + 256 * (a3 + 256 * (a4 +
                         256 * (a5 + 256 * (a6 + 256 * a7))))))

        alpha is a 64-bit unsigned integer, from which a four-bit alpha value
        is extracted for a texel at location (x,y) in the block using:

            alpha(x,y) = bits[4*(4*y+x)+3..4*(4*y+x)+0]

        where bit 63 is the most significant and bit 0 is the least
        significant bit.

        The alpha component for a texel at location (x,y) in the block is
        given by alpha(x,y) / 15.


    COMPRESSED_RGBA_S3TC_DXT5_EXT:  Each 4x4 block of texels consists of 64
    bits of compressed alpha image data followed by 64 bits of RGB image data.

    Each RGB image data block is encoded according to the
    COMPRESSED_RGB_S3TC_DXT1_EXT format, with the exception that the two code
    bits always use the non-transparent encodings.  In other words, they are
    treated as though color0 > color1, regardless of the actual values of
    color0 and color1.

    Each alpha image data block is encoded as a sequence of 8 bytes, called
    (in order of increasing address):

        alpha0, alpha1, bits_0, bits_1, bits_2, bits_3, bits_4, bits_5

        The alpha0 and alpha1 are 8-bit unsigned bytes converted to alpha
        components by multiplying by 1/255.

        The 6 "bits" bytes of the block are decoded into one 48-bit integer:

          bits = bits_0 + 256 * (bits_1 + 256 * (bits_2 + 256 * (bits_3 +
                          256 * (bits_4 + 256 * bits_5))))
        bits is a 48-bit unsigned integer, from which a three-bit control code
        is extracted for a texel at location (x,y) in the block using:

            code(x,y) = bits[3*(4*y+x)+1..3*(4*y+x)+0]

        where bit 47 is the most significant and bit 0 is the least
        significant bit.

        The alpha component for a texel at location (x,y) in the block is
        given by:

              alpha0,                   code(x,y) == 0
              alpha1,                   code(x,y) == 1

              (6*alpha0 + 1*alpha1)/7,  alpha0 > alpha1 and code(x,y) == 2
              (5*alpha0 + 2*alpha1)/7,  alpha0 > alpha1 and code(x,y) == 3
              (4*alpha0 + 3*alpha1)/7,  alpha0 > alpha1 and code(x,y) == 4
              (3*alpha0 + 4*alpha1)/7,  alpha0 > alpha1 and code(x,y) == 5
              (2*alpha0 + 5*alpha1)/7,  alpha0 > alpha1 and code(x,y) == 6
              (1*alpha0 + 6*alpha1)/7,  alpha0 > alpha1 and code(x,y) == 7

              (4*alpha0 + 1*alpha1)/5,  alpha0 <= alpha1 and code(x,y) == 2
              (3*alpha0 + 2*alpha1)/5,  alpha0 <= alpha1 and code(x,y) == 3
              (2*alpha0 + 3*alpha1)/5,  alpha0 <= alpha1 and code(x,y) == 4
              (1*alpha0 + 4*alpha1)/5,  alpha0 <= alpha1 and code(x,y) == 5
              0.0,                      alpha0 <= alpha1 and code(x,y) == 6
              1.0,                      alpha0 <= alpha1 and code(x,y) == 7
```

NVIDIA Implementation Note

NVIDIA GeForce 6 and 7 Series of GPUs (NV4x- and G7x-based GPUs) and their Quadro counterparts (Quadro FX 4000, 4400, 4500; Quadro NVS 440; etc.) do not ignore the order of the 16-bit RGB values color0 and color1 when decoding DXT3 and DXT5 texture formats (i.e., COMPRESSED_RGBA_S3TC_DXT5_EXT and COMPRESSED_RGBA_S3TC_DXT5_EXT). This is at variance with the specification language saying:

> Each RGB image data block is encoded according to the COMPRESSED_RGB_S3TC_DXT1_EXT format, with the exception that the two code bits always use the non-transparent encodings. In other words, they are treated as though color0 > color1, regardless of the actual values of color0 and color1.

With these NV4x and G7x GPUs, when decoding the DXT3 and DXT5 formats, if color0 <= color1 then the code(x,y) values of 2 and 3 encode (RGB0+RGB1)/2 and BLACK respectively (as is the case for DXT1).

All other NVIDIA GPUs (those based on GPU designs other than NV4x and G7x) implement DXT3 and DXT5 decoding strictly according to the specification. Specifically, the order of color0 and color1 does not affect the decoding of the DXT3 and DXT5 format, consistent with the specification paragraph cited above.

To ensure reliable decoding of DXT3 and DXT5 textures, please avoid encoding an RGB image data block with color0 <= color1 when the block also uses code(x,y) values of 2 and 3.

Revision History

1.3	07/07/07 mjk	Correct NVIDIA note about DXT3/5 decoding issue.
1.2	01/26/06 mjk	Add NVIDIA note about DXT3/5 decoding issue.

```
    1.1,  11/16/01 pbrown:    Updated contact info, clarified where texels
                              fall within a single block.

    1.0,  07/07/00 prbrown1:  Published final version agreed to by working
                              group members.

    0.9,  06/24/00 prbrown1:  Documented that block-aligned TexSubImage calls
                              do not modify existing texels outside the
                              modified blocks.  Added caveat to allow for a
                              (0,0)-anchored TexSubImage operation of
                              arbitrary size.

    0.7,  04/11/00 prbrown1:  Added issues on DXT1, DXT3, and DXT5 encodings
                              where the MSDN documentation doesn't match what
                              is really done.  Added enum values from the
                              extension registry.

    0.4,  03/28/00 prbrown1:  Updated to reflect final version of the
                              ARB_texture_compression extension.  Allowed
                              block-aligned TexSubImage calls.

    0.3,  03/07/00 prbrown1:  Resolved issues pertaining to the format of RGB
                              blocks in the DXT3 and DXT5 formats (they don't
                              ever use the "transparent" encoding).  Fixed
                              decoding of DXT1 blocks.  Pointed out issue of
                              "transparent" texels in DXT1 encodings having
                              different behaviors for RGB and RGBA internal
                              formats.

    0.2,  02/23/00 prbrown1:  Minor revisions; added several issues.

    0.11, 02/17/00 prbrown1:  Slight modification to error semantics
      (INVALID_ENUM instead of INVALID_OPERATION).

    0.1,  02/15/00 prbrown1:  Initial revisio
```

Name

EXT_texture_cube_map

Name Strings

GL_EXT_texture_cube_map

Forward Compatibility

This extension is superceded by the ARB_texture_cube_map extension that is officially sanctioned by the OpenGL Architectural Review Board. Enumerant values for EXT_texture_cube_map and ARB_texture_cube_map are identical. The two extensions are operationally identical; the only difference is the change of identifier from EXT to ARB.

Because the enumerants are identical for the two extensions and because there are no new entry points, an application that detects either the "GL_EXT_texture_cube_map" or "GL_ARB_texture_cube_map" extension name will operate correctly using either extension.

NVIDIA's Release 4 drivers and early versions of NVIDIA's Release 5 drivers advertised the EXT_texture_cube_map without also advertising the ARB_texture_cube_map extension because the ARB version of the extension was not then available. To ensure that your applications operate correctly with these older drivers, NVIDIA recommends that you query for either the EXT_texture_cube_map or ARB_texture_cube_map extension to determine when texture cube map functionality is available. Because the enumerants and functionality is unchanged, programs written to use ARB_texture_cube_map need only recognize EXT_texture_cube_map to operate correctly.

Name

    EXT_texture_edge_clamp

Name Strings

    GL_EXT_texture_edge_clamp

Version

    $Date: 1997/09/22 23:04:01 $ $Revision: 1.1 $

Dependencies

    SGIS_texture_filter4 affects the definition of this extension

Overview

    The base OpenGL provides clamping such that the texture coordinates are
    limited to exactly the range [0,1]. When a texture coordinate is
    clamped using this algorithm, the texture sampling filter straddles the
    edge of the texture image, taking 1/2 its sample values from within the
    texture image, and the other 1/2 from the texture border. It is
    sometimes desirable to clamp a texture without requiring a border, and
    without using the constant border color.

    This extension defines a new texture clamping algorithm.
    CLAMP_TO_EDGE_EXT clamps texture coordinates at all mipmap levels such
    that the texture filter never samples a border texel. When used with a
    NEAREST or a LINEAR filter, the color returned when clamping is derived
    only from texels at the edge of the texture image. When used with
    FILTER4 filters, the filter operations of CLAMP_TO_EDGE_EXT are defined
    but don't result in a nice clamp-to-edge color.

    CLAMP_TO_EDGE_EXT is supported by 1, 2, and 3-dimensional textures
    only.

Issues

    *   Is the arithmetic for FILTER4 filters correct? Is this the right
        thing to do?

New Procedures and Functions

    None

New Tokens

    Accepted by the <param> parameter of TexParameteri and TexParameterf,
    and by the <params> parameter of TexParameteriv and TexParameterfv, when
    their <pname> parameter is TEXTURE_WRAP_S, TEXTURE_WRAP_T, or
    TEXTURE_WRAP_R:

        CLAMP_TO_EDGE_EXT   0x812F

Additions to Chapter 2 of the 1.0 Specification (OpenGL Operation)

    None

Additions to Chapter 3 of the 1.0 Specification (Rasterization)

GL Specification Table 3.7 is updated as follows:

Name	Type	Legal Values
TEXTURE_WRAP_S	integer	CLAMP, REPEAT, CLAMP_TO_EDGE_EXT
TEXTURE_WRAP_T	integer	CLAMP, REPEAT, CLAMP_TO_EDGE_EXT
TEXTURE_WRAP_R	integer	CLAMP, REPEAT, CLAMP_TO_EDGE_EXT
TEXTURE_MIN_FILTER	integer	NEAREST, LINEAR, NEAREST_MIPMAP_NEAREST, NEAREST_MIPMAP_LINEAR, LINEAR_MIPMAP_NEAREST, LINEAR_MIPMAP_LINEAR, FILTER4_SGIS, LINEAR_CLIPMAP_LINEAR_SGIX
TEXTURE_MAG_FILTER	integer	NEAREST, LINEAR, FILTER4_SGIS, LINEAR_DETAIL_SGIS, LINEAR_DETAIL_ALPHA_SGIS, LINEAR_DETAIL_COLOR_SGIS, LINEAR_SHARPEN_SGIS, LINEAR_SHARPEN_ALPHA_SGIS, LINEAR_SHARPEN_COLOR_SGIS, LINEAR_LEQUAL_R_SGIS, LINEAR_GEQUAL_R_SGIS
TEXTURE_BORDER_COLOR	4 floats	any 4 values in [0,1]
DETAIL_TEXTURE_LEVEL_SGIS	integer	any non-negative integer
DETAIL_TEXTURE_MODE_SGIS	integer	ADD, MODULATE
TEXTURE_MIN_LOD	float	any value
TEXTURE_MAX_LOD	float	any value
TEXTURE_BASE_LEVEL	integer	any non-negative integer
TEXTURE_MAX_LEVEL	integer	any non-negative integer
GENERATE_MIPMAP_SGIS	boolean	TRUE or FALSE
TEXTURE_CLIPMAP_OFFSET_SGIX	2 floats	any 2 values

Table 3.7: Texture parameters and their values.

CLAMP_TO_EDGE_EXT texture clamping is specified by calling TexParameteri with <target> set to TEXTURE_1D, TEXTURE_2D, or TEXTURE_3D, <pname> set to TEXTURE_WRAP_S, TEXTURE_WRAP_T, or TEXTURE_WRAP_R, and <param> set to CLAMP_TO_EDGE_EXT.

Let [min,max] be the range of a clamped texture coordinate, and let N be the size of the 1D, 2D, or 3D texture image in the direction of clamping. Then in all cases

```
    max = 1 - min
```

because the clamping is always symmetric about the [0,1] mapped range of a texture coordinate. When used with NEAREST or LINEAR filters, CLAMP_TO_EDGE_EXT defines a minimum clamping value of

```
    min = 1 / 2*N
```

When used with FILTER4 filters, CLAMP_TO_EDGE_EXT defines a minimum clamping value of

```
    min = 3 / 2*N,          N > 2

    min = 1/2               N <= 2
```

Additions to Chapter 4 of the 1.0 Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the 1.0 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.0 Specification (State and State Requests)

None

Additions to the GLX Specification

None

Dependencies on SGIS_texture_filter4

If SGIS_texture_filter4 is not implemented, then discussions about the interaction of filter4 texture filters and the clamping function described in this file are invalid, and should be ignored.

Errors

None

New State

Only the type information changes for these parameters:

Get Value	Get Command	Type	Initial Value	Attrib
TEXTURE_WRAP_S	GetTexParameteriv	n x Z3	REPEAT	texture
TEXTURE_WRAP_T	GetTexParameteriv	n x Z3	REPEAT	texture
TEXTURE_WRAP_R	GetTexParameteriv	n x Z3	REPEAT	texture

New Implementation Dependent State

None

Name

    EXT_texture_env_add

Name Strings

    GL_EXT_texture_env_add

Status

    Shipping (version 1.6)

Version

    $Date: 1999/03/22 17:28:00 $ $Revision: 1.1 $

Number

    185

Dependencies

    None

Overview

```
    New texture environment function ADD is supported with the following
    equation:
                        Cv = Cf + Ct

    New function may be specified by calling TexEnv with ADD token.
```

New Procedures and Functions

    None

New Tokens

```
    Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv, and
    TexEnvfi when the <pname> parameter value is GL_TEXTURE_ENV_MODE

        ADD
```

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

        None

Additions to Chapter 3 of the GL Specification (Rasterization)

	Texture Environment				
Base Texture Format	REPLACE	MODULATE	BLEND	DECAL	ADD
ALPHA	...	...	...	...	Rv = Rf
	...	...	...	...	Gv = Gf
	...	...	...	...	Bv = Bf
	...	...	...	...	Av = AfAt
LUMINANCE	...	...	...	...	Rv = Rf+Lt
	...	...	...	...	Gv = Gf+Lt
	...	...	...	...	Bv = Bf+Lt
	...	...	...	...	Av = Af
LUMINANCE_ALPHA	...	...	...	...	Rv = Rf+Lt
	...	...	...	...	Gv = Gf+Lt
	...	...	...	...	Bv = Bf+Lt
	...	...	...	...	Av = AfAt
INTENSITY	...	...	...	...	Rv = Rf+It
	...	...	...	...	Gv = Gf+It
	...	...	...	...	Bv = Bf+It
	...	...	...	...	Av = Af+It
RGB	...	...	...	...	Rv = Rf+Rt
	...	...	...	...	Gv = Gf+Gt
	...	...	...	...	Bv = Bf+Bt
	...	...	...	...	Av = Af
RGBA	...	...	...	...	Rv = Rf+Rt
	...	...	...	...	Gv = Gf+Gt
	...	...	...	...	Bv = Bf+Bt
	...	...	...	...	Av = AfAt

Table 3.11: Texture functions.

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the GL Specification (Special Functions)

None

Additions to Chapter 6 of the GL Specification (State and State Requests)

None

Additions to the GLX / WGL / AGL Specifications

None

GLX Protocol

None

Errors

None

New State

None

New Implementation Dependent State

None

Name

    EXT_texture_env_combine

Name Strings

    GL_EXT_texture_env_combine

Version

    $Date: 1999/04/02 13:54:17 $ $Revision: 1.7 $

Number

    158

Dependencies

    SGI_texture_color_table affects the definition of this extension
    SGIX_texture_scale_bias affects the definition of this extension

Overview

```
    New texture environment function COMBINE_EXT allows programmable
    texture combiner operations, including:

        REPLACE             Arg0
        MODULATE            Arg0 * Arg1
        ADD                 Arg0 + Arg1
        ADD_SIGNED_EXT      Arg0 + Arg1 - 0.5
        INTERPOLATE_EXT     Arg0 * (Arg2) + Arg1 * (1-Arg2)

    where Arg0, Arg1 and Arg2 are derived from

     PRIMARY_COLOR_EXT    primary color of incoming fragment
     TEXTURE              texture color of corresponding texture unit
     CONSTANT_EXT         texture environment constant color
     PREVIOUS_EXT         result of previous texture environment; on
                          texture unit 0, this maps to PRIMARY_COLOR_EXT

    and Arg2 is restricted to the alpha component of the corresponding source.

    In addition, the result may be scaled by 1.0, 2.0 or 4.0.
```

Issues

```
    Should the explicit bias be removed in favor of an implcit bias as
    part of a ADD_SIGNED_EXT function?

     - Yes.  This pre-scale bias is a special case and will be treated
       as such.

    Should the primary color of the incoming fragment be available to
    all texture environments?  Currently it is only available to the
    texture environment of texture unit 0.

     - Yes, PRIMARY_COLOR_EXT has been added as an input source.
    Should textures from other texture units be allowed as sources?

      - No, not in the base spec.  Too many vendors have expressed
        concerns about the scalability of such functionality.  This can
        be added as a subsequent extension.

    All of the 1.2 modes except BLEND can be expressed in terms of
    this extension.  Should texture color be allowed as a source for
    Arg2, so all of the 1.2 modes can be expressed?  If so, should all
    color sources be allowed, to maintain orthogonality?

      - No, not in the base spec.  This can be added as a subsequent
        extension.
```

New Procedures and Functions

```
    None
```

New Tokens

```
    Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv,
    and TexEnviv when the <pname> parameter value is TEXTURE_ENV_MODE

        COMBINE_EXT                                     0x8570

    Accepted by the <pname> parameter of TexEnvf, TexEnvi, TexEnvfv,
    and TexEnviv when the <target> parameter value is TEXTURE_ENV

        COMBINE_RGB_EXT                                 0x8571
        COMBINE_ALPHA_EXT                               0x8572
        SOURCE0_RGB_EXT                                 0x8580
        SOURCE1_RGB_EXT                                 0x8581
        SOURCE2_RGB_EXT                                 0x8582
        SOURCE0_ALPHA_EXT                               0x8588
        SOURCE1_ALPHA_EXT                               0x8589
        SOURCE2_ALPHA_EXT                               0x858A
        OPERAND0_RGB_EXT                                0x8590
        OPERAND1_RGB_EXT                                0x8591
        OPERAND2_RGB_EXT                                0x8592
        OPERAND0_ALPHA_EXT                              0x8598
        OPERAND1_ALPHA_EXT                              0x8599
        OPERAND2_ALPHA_EXT                              0x859A
        RGB_SCALE_EXT                                   0x8573
        ALPHA_SCALE

    Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv,
    and TexEnviv when the <pname> parameter value is COMBINE_RGB_EXT
    or COMBINE_ALPHA_EXT

        REPLACE
        MODULATE
        ADD
        ADD_SIGNED_EXT                                  0x8574
        INTERPOLATE_EXT                                 0x8575
    Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv,
    and TexEnviv when the <pname> parameter value is SOURCE0_RGB_EXT,
    SOURCE1_RGB_EXT, SOURCE2_RGB_EXT, SOURCE0_ALPHA_EXT,
    SOURCE1_ALPHA_EXT, or SOURCE2_ALPHA_EXT

        TEXTURE
        CONSTANT_EXT                                    0x8576
        PRIMARY_COLOR_EXT                               0x8577
        PREVIOUS_EXT                                    0x8578

    Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv,
    and TexEnviv when the <pname> parameter value is
    OPERAND0_RGB_EXT or OPERAND1_RGB_EXT

        SRC_COLOR
        ONE_MINUS_SRC_COLOR
        SRC_ALPHA
        ONE_MINUS_SRC_ALPHA

    Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv,
    and TexEnviv when the <pname> parameter value is
    OPERAND0_ALPHA_EXT or OPERAND1_ALPHA_EXT

        SRC_ALPHA
        ONE_MINUS_SRC_ALPHA

    Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv,
    and TexEnviv when the <pname> parameter value is
    OPERAND2_RGB_EXT or OPERAND2_ALPHA_EXT

        SRC_ALPHA

    Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv,
    and TexEnviv when the <pname> parameter value is RGB_SCALE_EXT or
    ALPHA_SCALE

        1.0
        2.0
        4.0
```

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

    None

Additions to Chapter 3 of the GL Specification (Rasterization)

```
    Added to subsection 3.8.9, before the paragraph describing the
    state requirements:

    If the value of TEXTURE_ENV_MODE is COMBINE_EXT, the form of the
    texture function depends on the values of COMBINE_RGB_EXT and
    COMBINE_ALPHA_EXT, according to table 3.20.  The RGB and ALPHA
    results of the texture function are then multiplied by the values
    of RGB_SCALE_EXT and ALPHA_SCALE, respectively.  The results are
    clamped to [0,1].
```

COMBINE_RGB_EXT or COMBINE_ALPHA_EXT	Texture Function
REPLACE	Arg0
MODULATE	Arg0 * Arg1
ADD	Arg0 + Arg1
ADD_SIGNED_EXT	Arg0 + Arg1 - 0.5
INTERPOLATE_EXT	Arg0 * (Arg2) + Arg1 * (1-Arg2)

Table 3.20: COMBINE_EXT texture functions

The arguments Arg0, Arg1 and Arg2 are determined by the values of SOURCE<n>_RGB_EXT, SOURCE<n>_ALPHA_EXT, OPERAND<n>_RGB_EXT and OPERAND<n>_ALPHA_EXT. In the following two tables, Ct and At are the filtered texture RGB and alpha values; Cc and Ac are the texture environment RGB and alpha values; Cf and Af are the RGB and alpha of the primary color of the incoming fragment; and Cp and Ap are the RGB and alpha values resulting from the previous texture environment. On texture environment 0, Cp and Ap are identical to Cf and Af, respectively. The relationship is described in tables 3.21 and 3.22.

SOURCE<n>_RGB_EXT	OPERAND<n>_RGB_EXT	Argument
TEXTURE	SRC_COLOR	Ct
	ONE_MINUS_SRC_COLOR	(1-Ct)
	SRC_ALPHA	At
	ONE_MINUS_SRC_ALPHA	(1-At)
CONSTANT_EXT	SRC_COLOR	Cc
	ONE_MINUS_SRC_COLOR	(1-Cc)
	SRC_ALPHA	Ac
	ONE_MINUS_SRC_ALPHA	(1-Ac)
PRIMARY_COLOR_EXT	SRC_COLOR	Cf
	ONE_MINUS_SRC_COLOR	(1-Cf)
	SRC_ALPHA	Af
	ONE_MINUS_SRC_ALPHA	(1-Af)
PREVIOUS_EXT	SRC_COLOR	Cp
	ONE_MINUS_SRC_COLOR	(1-Cp)
	SRC_ALPHA	Ap
	ONE_MINUS_SRC_ALPHA	(1-Ap)

Table 3.21: Arguments for COMBINE_RGB_EXT functions

SOURCE<n>_ALPHA_EXT	OPERAND<n>_ALPHA_EXT	Argument
TEXTURE	SRC_ALPHA	At
	ONE_MINUS_SRC_ALPHA	(1-At)
CONSTANT_EXT	SRC_ALPHA	Ac
	ONE_MINUS_SRC_ALPHA	(1-Ac)
PRIMARY_COLOR_EXT	SRC_ALPHA	Af
	ONE_MINUS_SRC_ALPHA	(1-Af)
PREVIOUS_EXT	SRC_ALPHA	Ap
	ONE_MINUS_SRC_ALPHA	(1-Ap)

Table 3.22: Arguments for COMBINE_ALPHA_EXT functions

The mapping of texture components to source components is summarized in Table 3.23. In the following table, At, Lt, It, Rt, Gt and Bt are the filtered texel values.

Base Internal Format	RGB Values	Alpha Value
ALPHA	0, 0, 0	At
LUMINANCE	Lt, Lt, Lt	1
LUMINANCE_ALPHA	Lt, Lt, Lt	At
INTENSITY	It, It, It	It
RGB	Rt, Gt, Bt	1
RGBA	Rt, Gt, Bt	At

Table 3.23: Correspondence of texture components to source components for COMBINE_RGB_EXT and COMBINE_ALPHA_EXT arguments

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the GL Specification (Special Functions)

None

Additions to Chapter 6 of the GL Specification (State and State Requests)

None

Additions to the GLX Specification

None

GLX Protocol

None

Errors

INVALID_ENUM is generated if <params> value for COMBINE_RGB_EXT or COMBINE_ALPHA_EXT is not one of REPLACE, MODULATE, ADD, ADD_SIGNED_EXT, or INTERPOLATE_EXT.

INVALID_ENUM is generated if <params> value for SOURCE0_RGB_EXT, SOURCE1_RGB_EXT, SOURCE2_RGB_EXT, SOURCE0_ALPHA_EXT, SOURCE1_ALPHA_EXT or SOURCE2_ALPHA_EXT is not one of TEXTURE, CONSTANT_EXT, PRIMARY_COLOR_EXT or PREVIOUS_EXT.

INVALID_ENUM is generated if <params> value for OPERAND0_RGB_EXT or OPERAND1_RGB_EXT is not one of SRC_COLOR, ONE_MINUS_SRC_COLOR, SRC_ALPHA or ONE_MINUS_SRC_ALPHA.

INVALID_ENUM is generated if <params> value for OPERAND0_ALPHA_EXT or OPERAND1_ALPHA_EXT is not one of SRC_ALPHA or ONE_MINUS_SRC_ALPHA.

INVALID_ENUM is generated if <params> value for OPERAND2_RGB_EXT or OPERAND2_ALPHA_EXT is not SRC_ALPHA.

INVALID_VALUE is generated if <params> value for RGB_SCALE_EXT or ALPHA_SCALE is not one of 1.0, 2.0, or 4.0.

Dependencies on SGI_texture_color_table

If SGI_texture_color_table is implemented, the expanded Rt, Gt, Bt, and At values are used directly instead of the expansion described by Table 3.23.

Dependencies on SGIX_texture_scale_bias

If SGIX_texture_scale_bias is implemented, the expanded Rt, Gt, Bt, and At values are used directly instead of the expansion described by Table 3.23.

New State

Get Value	Get Command	Type	Initial Value	Attribute
COMBINE_RGB_EXT	GetTexEnviv	n x Z4	MODULATE	texture
COMBINE_ALPHA_EXT	GetTexEnviv	n x Z4	MODULATE	texture
SOURCE0_RGB_EXT	GetTexEnviv	n x Z3	TEXTURE	texture
SOURCE1_RGB_EXT	GetTexEnviv	n x Z3	PREVIOUS_EXT	texture
SOURCE2_RGB_EXT	GetTexEnviv	n x Z3	CONSTANT_EXT	texture
SOURCE0_ALPHA_EXT	GetTexEnviv	n x Z3	TEXTURE	texture
SOURCE1_ALPHA_EXT	GetTexEnviv	n x Z3	PREVIOUS_EXT	texture
SOURCE2_ALPHA_EXT	GetTexEnviv	n x Z3	CONSTANT_EXT	texture
OPERAND0_RGB_EXT	GetTexEnviv	n x Z6	SRC_COLOR	texture
OPERAND1_RGB_EXT	GetTexEnviv	n x Z6	SRC_COLOR	texture
OPERAND2_RGB_EXT	GetTexEnviv	n x Z1	SRC_ALPHA	texture
OPERAND0_ALPHA_EXT	GetTexEnviv	n x Z4	SRC_ALPHA	texture
OPERAND1_ALPHA_EXT	GetTexEnviv	n x Z4	SRC_ALPHA	texture
OPERAND2_ALPHA_EXT	GetTexEnviv	n x Z1	SRC_ALPHA	texture
RGB_SCALE_EXT	GetTexEnvfv	n x R3	1.0	texture
ALPHA_SCALE	GetTexEnvfv	n x R3	1.0	texture

New Implementation Dependent State

None

NVIDIA Implementation Details

Because of a hardware limitation, TNT, TNT2, GeForce, and Quadro treat "scale by 4.0" with the COMBINE_RGB_EXT or COMBINE_ALPHA_EXT mode of ADD_SIGNED_EXT as "scale by 2.0".

Name

    EXT_texture_env_dot3

Name Strings

    EXT_texture_env_dot3

Notice

IP Status

    None

Version

    $Date: 2000/09/28 13:54:17 $ $Revision: 1.2 $

Number

    None.

Dependencies

```
    EXT_texture_env_combine is required and is modified by this extension
    ARB_multitexture affects the definition of this extension
```

Overview

```
    Adds new operation to the texture combiner operations.

        DOT3_RGB_EXT                    Arg0 <dotprod> Arg1
        DOT3_RGBA_EXT                   Arg0 <dotprod> Arg1

    where Arg0, Arg1 are derived from

        PRIMARY_COLOR_EXT       primary color of incoming fragment
        TEXTURE                 texture color of corresponding texture unit
        CONSTANT_EXT            texture environment constant color
        PREVIOUS_EXT            result of previous texture environment; on
                                texture unit 0, this maps to PRIMARY_COLOR_EXT

    This operaion can only be performed if SOURCE0_RGB_EXT,
    SOURCE1_RGB_EXT are defined.
```

Issues

    None

New Procedures and Functions

    None

New Tokens

Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv, and TexEnviv when the <pname> parameter value is COMBINE_RGB_EXT

DOT3_RGB_EXT	0x8740
DOT3_RGBA_EXT	0x8741

Additions to Chapter 2 of the OpenGL 1.2 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the OpenGL 1.2 Specification (Rasterization)

Added to subsection 3.8.9, before the paragraph describing the state requirements:

If the value of TEXTURE_ENV_MODE is COMBINE_EXT, the form of the texture function depends on the values of COMBINE_RGB_EXT and COMBINE_ALPHA_EXT, according to table 3.20. The RGB and ALPHA results of the texture function are not multiplied by the values of RGB_SCALE_EXT and ALPHA_SCALE, respectively. The results are clamped to [0,1].

```
    COMBINE_RGB_EXT         Texture Function
    ------------------      ----------------
    DOT3_RGB_EXT            4*((Arg0_r - 0.5)*(Arg1_r - 0.5) +
                               (Arg0_g - 0.5)*(Arg1_g - 0.5) +
                               (Arg0_b - 0.5)*(Arg1_b - 0.5))
                            This value is placed into all three
                            r,g,b components of the output.
    DOT3_RGBA_EXT           4*((Arg0_r - 0.5)*(Arg1_r - 0.5) +
                               (Arg0_g - 0.5)*(Arg1_g - 0.5) +
                               (Arg0_b - 0.5)*(Arg1_b - 0.5))
                            This value is placed into all four
                            r,g,b,a components of the output.
```

Table 3.20: COMBINE_EXT texture functions

Additions to Chapter 4 of the OpenGL 1.2 Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the OpenGL 1.2 Specification (Special Functions)

None

Additions to Chapter 6 of the OpenGL 1.2 Specification (State and State Requests)

None

Additions to the AGL/GLX/WGL Specifications

    None

GLX Protocol

    None

Errors

Modifications to EXT_texture_env_combine

Dependencies on ARB_multitexture

New State

    None

New Implementation Dependent State

    None

Revision History

    None

Name

EXT_texture_filter_anisotropic

Name Strings

GL_EXT_texture_filter_anisotropic

Notice

Version

August 24, 1999

Number

187

Dependencies

Written based on the wording of the OpenGL 1.2 specification.

Overview

Texture mapping using OpenGL's existing mipmap texture filtering modes assumes that the projection of the pixel filter footprint into texture space is a square (ie, isotropic). In practice however, the footprint may be long and narrow (ie, anisotropic). Consequently, mipmap filtering severely blurs images on surfaces angled obliquely away from the viewer.

Several approaches exist for improving texture sampling by accounting for the anisotropic nature of the pixel filter footprint into texture space. This extension provides a general mechanism for supporting anisotropic texturing filtering schemes without specifying a particular formulation of anisotropic filtering.

The extension permits the OpenGL application to specify on a per-texture object basis the maximum degree of anisotropy to account for in texture filtering.

Increasing a texture object's maximum degree of anisotropy may improve texture filtering but may also significantly reduce the implementation's texture filtering rate. Implementations are free to clamp the specified degree of anisotropy to the implementation's maximum supported degree of anisotropy.

A texture's maximum degree of anisotropy is specified independent from the texture's minification and magnification filter (as opposed to being supported as an entirely new filtering mode). Implementations are free to use the specified minification and magnification filter to select a particular anisotropic texture filtering scheme. For example, a NEAREST filter with a maximum degree of anisotropy of two could be treated as a 2-tap filter that accounts for the direction of anisotropy. Implementations are also permitted to ignore the minification or magnification filter and implement the highest quality of anisotropic filtering possible.

Applications seeking the highest quality anisotropic filtering available are advised to request a LINEAR_MIPMAP_LINEAR minification filter, a LINEAR magnification filter, and a large maximum degree of anisotropy.

Issues

Should there be a particular anisotropic texture filtering minification and magnification mode?

RESOLUTION: NO. The maximum degree of anisotropy should control when anisotropic texturing is used. Making this orthogonal to the minification and magnification filtering modes allows these settings to influence the anisotropic scheme used. Yes, such an anisotropic filtering scheme exists in hardware.

What should the minimum value for MAX_TEXTURE_MAX_ANISTROPY_EXT be?

RESOLUTION: 2.0. To support this extension, at least 2 to 1 anisotropy should be supported.

Should an implementation-defined limit for the maximum maximum degree of anisotropy be "get-able"?

RESOLUTION: YES. But you should not assume that a high maximum maximum degree of anisotropy implies anything about texture filtering performance or quality.

Should anything particular be said about anisotropic 3D texture filtering?

Not sure. Does the implementation example shown in the spec for 2D anisotropic texture filtering readily extend to 3D anisotropic texture filtering?

New Procedures and Functions

None

New Tokens

Accepted by the <pname> parameters of GetTexParameterfv, GetTexParameteriv, TexParameterf, TexParameterfv, TexParameteri, and TexParameteriv:

```
TEXTURE_MAX_ANISOTROPY_EXT          0x84FE
```

Accepted by the <pname> parameters of GetBooleanv, GetDoublev, GetFloatv, and GetIntegerv:

```
MAX_TEXTURE_MAX_ANISOTROPY_EXT      0x84FF
```

Additions to Chapter 2 of the 1.2 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.2 Specification (Rasterization)

-- Sections 3.8.3 "Texture Parameters"

Add the following entry to the end of Table 3.17:

Name	Type	Legal Values
TEXTURE_MAX_ANISOTROPY_EXT	float	greater or equal to 1.0

-- Sections 3.8.5 "Texture Minification" and 3.8.6 "Texture Magnification"

After the first paragraph in Section 3.8.5:

"When the texture's value of TEXTURE_MAX_ANISOTROPY_EXT is equal to 1.0, the GL uses an isotropic texture filtering approach as described in this section and Section 3.8.6. However, when the texture's value of TEXTURE_MAX_ANISOTROPY_EXT is greater than 1.0, the GL implementation should use a texture filtering scheme that accounts for a degree of anisotropy up to the smaller of the value of TEXTURE_MAX_ANISTROPY_EXT or the implementation-defined value of MAX_TEXTURE_MAX_ANISTROPY_EXT.

The particular scheme for anisotropic texture filtering is implementation dependent. Additionally, implementations are free to consider the current texture minification and magnification modes to control the specifics of the anisotropic filtering scheme used.

The anisotropic texture filtering scheme may only access mipmap levels if the minification filter is one that requires mipmaps. Additionally, when a minification filter is specified, the anisotropic texture filtering scheme may only access texture mipmap levels between the texture's values for TEXTURE_BASE_LEVEL and TEXTURE_MAX_LEVEL, inclusive. Implementations are also recommended to respect the values of TEXTURE_MAX_LOD and TEXTURE_MIN_LOD to whatever extent the particular anisotropic texture filtering scheme permits this."

The following describes one particular approach to implementing anisotropic texture filtering for the 2D texturing case:

```
"Anisotropic texture filtering substantially changes Section 3.8.5.
Previously a single scale factor P was determined based on the
pixel's projection into texture space.  Now two scale factors,
Px and Py, are computed.

  Px = sqrt(dudx^2 + dvdx^2)
  Py = sqrt(dudy^2 + dvdy^2)

  Pmax = max(Px,Py)
  Pmin = min(Px,Py)

  N = min(ceil(Pmax/Pmin),maxAniso);
  Lamda' = log2(Pmax/N)

where maxAniso is the smaller of the texture's value of
TEXTURE_MAX_ANISOTROPY_EXT or the implementation-defined value of
MAX_TEXTURE_MAX_ANISOTROPY_EXT.

It is acceptable for implementation to round 'N' up to the nearest
supported sampling rate.  For example an implementation may only
support power-of-two sampling rates.

It is also acceptable for an implementation to approximate the ideal
functions Px and Py with functions Fx and Fy subject to the following
conditions:

  1.  Fx is continuous and monotonically increasing in |du/dx| and |dv/dx|.
      Fy is continuous and monotonically increasing in |du/dy| and |dv/dy|.

  2.  max(|du/dx|,|dv/dx|} <= Fx <= |du/dx| + |dv/dx|.
      max(|du/dy|,|dv/dy|} <= Fy <= |du/dy| + |dv/dy|.

Instead of a single sample, Tau, at (u,v,Lamda), 'N' locations in the
mipmap at LOD Lamda, are sampled within the texture footprint of the pixel.
This sum TauAniso is defined using the single sample Tau.  When the
texture's value of TEXTURE_MAX_ANISOTROPHY_EXT is greater than 1.0, use
TauAniso instead of Tau to determine the fragment's texture value.

                 i=N
                 ---
TauAniso = 1/N \ Tau(u(x - 1/2 + i/(N+1), y), v(x - 1/2 + i/(N+1), y)),  Px > Py
                 /
                 ---
                 i=1

                 i=N
                 ---
TauAniso = 1/N \ Tau(u(x, y - 1/2 + i/(N+1)), v(x, y - 1/2 + i/(N+1))),  Py >= Px
                 /
                 ---
                 i=1
```

It is acceptable to approximate the u and v functions with equally spaced samples in texture space at LOD Lamda:

```
                 i=N
                 ---
 TauAniso = 1/N  \ Tau(u(x,y)+dudx(i/(N+1)-1/2), v(x,y)+dvdx(i/(N+1)-1/2)), Px > Py
                 /
                 ---
                 i=1

                 i=N
                 ---
 TauAniso = 1/N  \ Tau(u(x,y)+dudy(i/(N+1)-1/2), v(x,y)+dvdy(i/(N+1)-1/2)), Py >= Px
                 /
                 ---
                 i=1
```

"

Additions to Chapter 4 of the 1.2 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.2 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

None

Additions to the GLX Specification

None

Errors

INVALID_VALUE is generated when TexParameter is called with <pname> of TEXTURE_MAX_ANISOTROPY_EXT and a <param> value or value of what <params> points to less than 1.0.

New State

(table 6.13, p203) add the entry:

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
TEXTURE_MAX_ANISOTROPY_EXT	R	GetTexParameterfv	1.0	Maximum degree of anisotropy	3.8.5	texture

New Implementation State

(table 6.25, p215) add the entry:

Get Value	Type	Get Command	Minimum Value	Description	Sec	Attribute
MAX_TEXTURE_MAX_ANISOTROPY_EXT	R	GetFloatv	2.0	Limit of maximum degree of anisotropy	3.8.5	-

Name

    EXT_texture_integer

Name Strings

    GL_EXT_texture_integer

Contact

    Michael Gold, NVIDIA Corporation (gold 'at' nvidia.com)
    Pat Brown, NVIDIA Corporation (pbrown 'at' nvidia.com)

Status

    Shipping for GeForce 8 Series (November 2006)

Version

```
    Last Modified Date:         07/15/2006
    NVIDIA Revision:            5
```

Number

    343

Dependencies

    OpenGL 2.0 is required.

    NV_gpu_program4 or EXT_gpu_shader4 is required.

    ARB_texture_float affects the definition of this extension.

    ARB_color_buffer_float affects the definition of this extension.

    EXT_framebuffer_object affects the definition of this extension.

    This extension is written against the OpenGL 2.0 specification.

Overview

    Fixed-point textures in unextended OpenGL have integer components,
    but those values are taken to represent floating-point values in
    the range [0,1]. These integer components are considered
    "normalized" integers. When such a texture is accessed by a
    shader or by fixed-function fragment processing, floating-point
    values are returned.

    This extension provides a set of new "unnormalized" integer texture
    formats. Formats with both signed and unsigned integers are provided. In
    these formats, the components are treated as true integers. When such
    textures are accessed by a shader, actual integer values are returned.

    Pixel operations that read from or write to a texture or color
    buffer with unnormalized integer components follow a path similar
    to that used for color index pixel operations, except that more

than one component may be provided at once. Integer values flow through the pixel processing pipe, and no pixel transfer operations are performed. Integer format enumerants used for such operations indicate unnormalized integer data.

Textures or render buffers with unnormalized integer formats may also be attached to framebuffer objects to receive fragment color values written by a fragment shader. Per-fragment operations that require floating-point color components, including multisample alpha operations, alpha test, blending, and dithering, have no effect when the corresponding colors are written to an integer color buffer. The NV_gpu_program4 and EXT_gpu_shader4 extensions add the capability to fragment programs and fragment shaders to write signed and unsigned integer output values.

This extension does not enforce type consistency for texture accesses or between fragment shaders and the corresponding framebuffer attachments. The results of a texture lookup from an integer texture are undefined:

* for fixed-function fragment processing, or

* for shader texture accesses expecting floating-point return values.

The color components used for per-fragment operations and written into a color buffer are undefined:

* for fixed-function fragment processing with an integer color buffer,

* for fragment shaders that write floating-point color components to an integer color buffer, or

* for fragment shaders that write integer color components to a color buffer with floating point or normalized integer components.

New Procedures and Functions

```
    void ClearColorIiEXT ( int r, int g, int b, int a );
    void ClearColorIuiEXT ( uint r, uint g, uint b, uint a );
    void TexParameterIivEXT( enum target, enum pname, int *params );
    void TexParameterIuivEXT( enum target, enum pname, uint *params );
    void GetTexParameterIivEXT ( enum target, enum pname, int *params);
    void GetTexParameterIuivEXT ( enum target, enum pname, uint *params);
```

New Tokens

Accepted by the <pname> parameters of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

```
        RGBA_INTEGER_MODE_EXT                           0x8D9E
```

Accepted by the <internalFormat> parameter of TexImage1D,
TexImage2D, and TexImage3D:

RGBA32UI_EXT	0x8D70
RGB32UI_EXT	0x8D71
ALPHA32UI_EXT	0x8D72
INTENSITY32UI_EXT	0x8D73
LUMINANCE32UI_EXT	0x8D74
LUMINANCE_ALPHA32UI_EXT	0x8D75
RGBA16UI_EXT	0x8D76
RGB16UI_EXT	0x8D77
ALPHA16UI_EXT	0x8D78
INTENSITY16UI_EXT	0x8D79
LUMINANCE16UI_EXT	0x8D7A
LUMINANCE_ALPHA16UI_EXT	0x8D7B
RGBA8UI_EXT	0x8D7C
RGB8UI_EXT	0x8D7D
ALPHA8UI_EXT	0x8D7E
INTENSITY8UI_EXT	0x8D7F
LUMINANCE8UI_EXT	0x8D80
LUMINANCE_ALPHA8UI_EXT	0x8D81
RGBA32I_EXT	0x8D82
RGB32I_EXT	0x8D83
ALPHA32I_EXT	0x8D84
INTENSITY32I_EXT	0x8D85
LUMINANCE32I_EXT	0x8D86
LUMINANCE_ALPHA32I_EXT	0x8D87
RGBA16I_EXT	0x8D88
RGB16I_EXT	0x8D89
ALPHA16I_EXT	0x8D8A
INTENSITY16I_EXT	0x8D8B
LUMINANCE16I_EXT	0x8D8C
LUMINANCE_ALPHA16I_EXT	0x8D8D
RGBA8I_EXT	0x8D8E
RGB8I_EXT	0x8D8F
ALPHA8I_EXT	0x8D90
INTENSITY8I_EXT	0x8D91
LUMINANCE8I_EXT	0x8D92
LUMINANCE_ALPHA8I_EXT	0x8D93

```
    Accepted by the <format> parameter of TexImage1D, TexImage2D,
    TexImage3D, TexSubImage1D, TexSubImage2D, TexSubImage3D,
    DrawPixels and ReadPixels:

        RED_INTEGER_EXT                                 0x8D94
        GREEN_INTEGER_EXT                               0x8D95
        BLUE_INTEGER_EXT                                0x8D96
        ALPHA_INTEGER_EXT                               0x8D97
        RGB_INTEGER_EXT                                 0x8D98
        RGBA_INTEGER_EXT                                0x8D99
        BGR_INTEGER_EXT                                 0x8D9A
        BGRA_INTEGER_EXT                                0x8D9B
        LUMINANCE_INTEGER_EXT                           0x8D9C
        LUMINANCE_ALPHA_INTEGER_EXT                     0x8D9D
```

Additions to Chapter 3 of the OpenGL 2.0 Specification (Rasterization)

Modify Section 3.6.4 (Rasterization of Pixel Rectangles), p. 126:

(modify the last paragraph, p. 126)
Pixels are drawn using

```
        void DrawPixels( sizei width, sizei height, enum format,
            enum type, void *data );
```

<format> is a symbolic constant indicating what the values in memory represent. <width> and <height> are the width and height, respectively, of the pixel rectangle to be drawn. <data> is a pointer to the data to be drawn. These data are represented with one of seven GL data types, specified by <type>. The correspondence between the twenty type token values and the GL data types they indicate is given in table 3.5. If the GL is in color index mode and <format> is not one of COLOR_INDEX, STENCIL_INDEX, or DEPTH_COMPONENT, then the error INVALID_OPERATION occurs. If the GL is in RGBA mode and the color buffer is an integer format and no fragment shader is active, the error INVALID_OPERATION occurs. If <type> is BITMAP and <format> is not COLOR_INDEX or STENCIL_INDEX then the error INVALID_ENUM occurs. If <format> is one of the integer component formats as defined in table 3.6, and <type> is FLOAT, then the error INVALID_ENUM occurs. Some additional constraints on the combinations of format and type values that are accepted is discussed below.

(add the following to table 3.6, p. 129)

Format Name	Element Meaning and Order	Target Buffer
RED_INTEGER_EXT	iR	Color
GREEN_INTEGER_EXT	iG	Color
BLUE_INTEGER_EXT	iB	Color
ALPHA_INTEGER_EXT	iA	Color
RGB_INTEGER_EXT	iR, iG, iB	Color
RGBA_INTEGER_EXT	iR, iG, iB, iA	Color
BGR_INTEGER_EXT	iB, iG, iR	Color
BGRA_INTEGER_EXT	iB, iG, iR, iA	Color
LUMINANCE_INTEGER_EXT	iLuminance	Color
LUMINANCE_ALPHA_INTEGER_EXT	iLuminance, iA	Color

Table 3.6: DrawPixels and ReadPixels formats. The second column gives a description of and the number and order of elements in a group. Unless specified as an index, formats yield components. Components are floating-point unless prefixed with the letter 'i' which indicates they are integer.

(modify first paragraph, p. 129)
Data are taken from host memory as a sequence of signed or unsigned bytes (GL data types byte and ubyte), signed or unsigned short integers (GL data types short and ushort), signed or unsigned integers (GL data types int and uint), or floating point values (GL data type float). These elements are grouped into sets of one, two, three, or four values, depending on the format, to form a group. Table 3.6 summarizes the format of groups obtained from memory; it also indicates those formats that yield indices and those that yield floating-point or integer components.

(modify the last paragraph, p. 135)
Conversion to floating-point

This step applies only to groups of floating-point components. It is not performed on indices or integer components.

(modify the third paragraph, p. 136)
Final Expansion to RGBA

This step is performed only for non-depth component groups. Each group is converted to a group of 4 elements as follows: if a group does not contain an A element, then A is added and set to 1 for integer components or 1.0 for floating-point components. If any of R, G, or B is missing from the group, each missing element is added and assigned a value of 0 for integer components or 0.0 for floating-point components.

(modify the last paragraph, p. 136)
Final Conversion

For a color index, final conversion consists of masking the bits of the index to the left of the binary point by 2^n - 1, where n is the number of bits in an index buffer. For floating-point RGBA components, each element is clamped to [0, 1]. The resulting values are converted to fixed-point according to the rules given in section 2.14.9 (Final Color Processing). For integer RGBA components, no conversion is applied. For a depth component, an element is first clamped to [0, 1] and then converted to fixed-point as if it were a window z value (see section 2.11.1, Controlling the Viewport). Stencil indices are masked by 2^n - 1, where n is the number of bits in the stencil buffer.

Modify Section 3.6.5 (Pixel Transfer Operations), p. 137

(modify last paragraph, p. 137)
The GL defines five kinds of pixel groups:

1. Floating-point RGBA component: Each group comprises four color components in floating point format: red, green, blue, and alpha.

2. Integer RGBA component: Each group comprises four color components in integer format: red, green, blue, and alpha.

3. Depth component: Each group comprises a single depth component.

4. Color index: Each group comprises a single color index.

5. Stencil index: Each group comprises a single stencil index.

(modify second paragraph, p. 138)
Each operation described in this section is applied sequentially to each pixel group in an image. Many operations are applied only to pixel groups of certain kinds; if an operation is not applicable to a given group, it is skipped. None of the operations defined in this section affect integer RGBA component pixel groups.

Modify Section 3.8 (Texturing), p. 149

(insert between the first and second paragraphs, p. 150)
The internal data type of a texture may be fixed-point, floating-point, signed integer or unsigned integer, depending on the internalformat of the texture. The correspondence between internalformat and the internal data type is given in table 3.16. Fixed-point and floating-point textures return a floating-point value and integer textures return signed or unsigned integer values. When a fragment shader is active, the shader is responsible for interpreting the result of a texture lookup as the correct data type, otherwise the result is undefined. Fixed functionality assumes floating-point data, hence the result of using fixed functionality with integer textures is undefined.

Modify Section 3.8.1 (Texture Image Specification), p. 150

(modify second paragraph, p. 151) The selected groups are processed exactly as for DrawPixels, stopping just before final conversion. If the <internalformat> of the texture is integer, the components are clamped to the representable range of the internal format: for signed formats, this is [-2^(n-1), 2^(n-1)-1] where n is the number of bits per component; for unsigned formats, the range is [0, 2^n-1]. For R, G, B, and A, if the

<internalformat> of the texture is fixed-point, the components are clamped to [0, 1]. Otherwise, the components are not modified.

(insert between paragraphs five and six, p. 151)
Textures with integer internal formats (table 3.16) require integer data. The error INVALID_OPERATION is generated if the internal format is integer and <format> is not one of the integer formats listed in table 3.6, or if the internal format is not integer and <format> is an integer format, or if <format> is an integer format and <type> is FLOAT.

(add the following to table 3.16, p. 154)

Sized Internal Format	Base Internal Format	R bits	G bits	B bits	A bits	L bits	I bits
ALPHA8I_EXT	ALPHA				i8		
ALPHA8UI_EXT	ALPHA				ui8		
ALPHA16I_EXT	ALPHA				i16		
ALPHA16UI_EXT	ALPHA				ui16		
ALPHA32I_EXT	ALPHA				i32		
ALPHA32UI_EXT	ALPHA				ui32		
LUMINANCE8I_EXT	LUMINANCE					i8	
LUMINANCE8UI_EXT	LUMINANCE					ui8	
LUMINANCE16I_EXT	LUMINANCE					i16	
LUMINANCE16UI_EXT	LUMINANCE					ui16	
LUMINANCE32I_EXT	LUMINANCE					i32	
LUMINANCE32UI_EXT	LUMINANCE					ui32	
LUMINANCE_ALPHA8I_EXT	LUMINANCE_ALPHA				i8	i8	
LUMINANCE_ALPHA8UI_EXT	LUMINANCE_ALPHA				ui8	ui8	
LUMINANCE_ALPHA16I_EXT	LUMINANCE_ALPHA				i16	i16	
LUMINANCE_ALPHA16UI_EXT	LUMINANCE_ALPHA				ui16	ui16	
LUMINANCE_ALPHA32I_EXT	LUMINANCE_ALPHA				i32	i32	
LUMINANCE_ALPHA32UI_EXT	LUMINANCE_ALPHA				ui32	ui32	
INTENSITY8I_EXT	INTENSITY						i8
INTENSITY8UI_EXT	INTENSITY						ui8
INTENSITY16I_EXT	INTENSITY						i16
INTENSITY16UI_EXT	INTENSITY						ui16
INTENSITY32I_EXT	INTENSITY						i32
INTENSITY32UI_EXT	INTENSITY						ui32
RGB8I_EXT	RGB	i8	i8	i8			
RGB8UI_EXT	RGB	ui8	ui8	ui8			
RGB16I_EXT	RGB	i16	i16	i16			
RGB16UI_EXT	RGB	ui16	ui16	ui16			
RGB32I_EXT	RGB	i32	i32	i32			
RGB32UI_EXT	RGB	ui32	ui32	ui32			
RGBA8I_EXT	RGBA	i8	i8	i8	i8		
RGBA8UI_EXT	RGBA	ui8	ui8	ui8	ui8		
RGBA16I_EXT	RGBA	i16	i16	i16	i16		
RGBA16UI_EXT	RGBA	ui16	ui16	ui16	ui16		
RGBA32I_EXT	RGBA	i32	i32	i32	i32		
RGBA32UI_EXT	RGBA	ui32	ui32	ui32	ui32		

Table 3.16: Correspondence of sized internal formats to base internal formats, internal data type and desired component resolutions for each sized internal format. The component resolution prefix indicates the internal data type: <f> is

floating point, <i> is signed integer, <ui> is unsigned integer, and no prefix is fixed-point.

Modify Section 3.8.2 (Alternate Texture Image Specification Commands), p. 159:

(modify the second paragraph, p. 159)
The error INVALID_OPERATION is generated if depth component data is required and no depth buffer is present, or if integer RGBA data is required and the format of the current color buffer is not integer, or if floating-point or fixed-point RGBA data is required and the format of the current color buffer is integer.

Modify Section 3.8.4 (Texture Parameters), p. 166:

Various parameters control how the texture array is treated when specified or changed, and when applied to a fragment. Each parameter is set by calling

```
    void TexParameter{if}( enum target, enum pname, T param );
    void TexParameter{if}v( enum target, enum pname, T params );
    void TexParameterIivEXT( enum target, enum pname, int *params );
    void TexParameterIuivEXT( enum target, enum pname, uint *params );
```

<target> is the target, either TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, or TEXTURE_CUBE_MAP. <pname> is a symbolic constant indicating the parameter to be set; the possible constants and corresponding parameters are summarized in table 3.19. In the first form of the command, <param> is a value to which to set a single-valued parameter; in the second and third forms of the command, <params> is an array of parameters whose type depends on the parameter being set.

If the value for TEXTURE_PRIORITY is specified as an integer, the conversion for signed integers from table 2.9 is applied to convert the value to floating-point. The floating point value of TEXTURE_PRIORITY is clamped to lie in [0, 1].

If the values for TEXTURE_BORDER_COLOR are specified with TexParameterIivEXT or TexParameterIuivEXT, the values are unmodified and stored with an internal data type of integer. If specified with TexParameteriv, the conversion for signed integers from table 2.9 is applied to convert these values to floating-point. Otherwise the values are unmodified and stored as floating-point.

(modify table 3.19, p. 167)

Name	Type	Legal Values
TEXTURE_BORDER_COLOR	4 floats or 4 ints or 4 uints	any 4 values

Table 3.19: Texture parameters and their values.

Modify Section 3.8.8 (Texture Minification), p. 170

(modify last paragraph, p. 174)

... If the texture contains color components, the values of TEXTURE_BORDER_COLOR are interpreted as an RGBA color to match the texture's internal format in a manner consistent with table 3.15. The internal data type of the border values must be consistent with the type returned by the texture as described in section 3.8, or the result is undefined. The border values for texture components stored as fixed-point values are clamped to [0, 1] before they are used. If the texture contains depth components, the first component of TEXTURE_BORDER_COLOR is interpreted as a depth value

Modify Section 3.8.10 (Texture Completeness), p. 177:

(add to the requirements for one-, two-, or three-dimensional textures)
If the internalformat is integer, TEXTURE_MAG_FILTER must be NEAREST and TEXTURE_MIN_FILTER must be NEAREST or NEAREST_MIPMAP_NEAREST.

Modify Section 3.11.2 (Shader Execution), p. 194

(modify Shader Outputs, first paragraph, p. 196)
... These are gl_FragColor, gl_FragData[n], and gl_FragDepth. If fragment clamping is enabled and the color buffer has a fixed-point or floating-point format, the final fragment color values or the final fragment data values written by a fragment shader are clamped to the range [0, 1]. If fragment clamping is disabled or the color buffer has an integer format, the final fragment color values or the final fragment data values are not modified. The final fragment depth...

(insert between the first paragraph and second paragraphs of "Shader Outputs", p. 196)
Colors values written by the fragment shader may be floating-point, signed integer or unsigned integer. If the color buffer has a fixed-point format, the color values are assumed to be floating-point and are converted to fixed-point as described in section 2.14.9; otherwise no type conversion is applied. If the values written by the fragment shader do not match the format(s) of the corresponding color buffer(s), the result is undefined.

Additions to Chapter 4 of the OpenGL 2.0 Specification (Per-Fragment Operations and the Frame Buffer)

Modify Chapter 4 Introduction, (p. 198)

(modify third paragraph, p. 198)
Color buffers consist of unsigned integer color indices, R, G, B and optionally A floating-point components represented as fixed-point unsigned integer or floating-point values, or R, G, B and optionally A integer components represented as signed or unsigned integer values. The number of bitplanes...

Modify Section 4.1.3 (Multisample Fragment Operations), p. 200

(modify the second paragraph in this section)
... If SAMPLE_ALPHA_TO_COVERAGE is enabled and the color buffer has a fixed-point or floating-point format, a temporary coverage value is generated ...

Modify Section 4.1.4 (Alpha Test), p. 201

(modify the first paragraph in this section)
This step applies only in RGBA mode and only if the color buffer has a fixed-point or floating-point format. In color index mode or if the color buffer has an integer format, proceed to the next operation. The alpha test discards ...

Modify Section 4.1.8 (Blending), p. 205

(modify the second paragraph, p. 206)
... Blending is dependent on the incoming fragment's alpha value and that of the corresponding currently stored pixel. Blending applies only in RGBA mode and only if the color buffer has a fixed-point or floating-point format; in color index mode or if the color buffer has an integer format, it is bypassed. ...

Modify Section 4.2.3 (Clearing the Buffers), p. 215

```
   void ClearColor(float r, float g, float b, float a);
```

sets the clear value for fixed-point and floating-point color buffers in RGBA mode. The specified components are stored as floating-point values.

```
   void ClearColorIiEXT(int r, int g, int b, int a);
   void ClearColorIuiEXT(uint r, uint g, uint b, uint a);
```

set the clear value for signed integer and unsigned integer color buffers, respectively, in RGBA mode. The specified components are stored as integer values.

(add to the end of first partial paragraph, p. 217) ... then a Clear directed at that buffer has no effect. When fixed-point RGBA color buffers are cleared, the clear color values are assumed to be floating-point and are clamped to [0,1] before being converted to fixed-point according to the rules of section 2.14.9. The result of clearing fixed-point or floating-point color buffers is undefined if the clear color was specified as integer values. The result of when clearing integer color buffers is undefined if the clear color was specified as floating-point values.

Modify Section 4.3.2 (Reading Pixels), p. 219

(append to the last paragraph, p. 221)
The error INVALID_OPERATION occurs if <format> is an integer format and the color buffer is not an integer format, or if the color buffer is an integer format and <format> is not. The error INVALID_ENUM occurs if <format> is an integer format and <type> is FLOAT.

(modify the first paragraph, p. 222)
... For a fixed-point color buffer, each element is taken to be a fixed-point value in [0, 1] with m bits, where m is the number of bits in the corresponding color component of the selected buffer (see section 2.14.9). For an integer or floating-point color buffer, the elements are unmodified.

(modify the section labeled "Conversion to L", p. 222)
This step applies only to RGBA component groups. If the format is either LUMINANCE or LUMINANCE_ALPHA, a value L is computed as

```
    L = R + G + B
```

otherwise if the format is either LUMINANCE_INTEGER_EXT or LUMINANCE_ALPHA_INTEGER_EXT, L is computed as

```
    L = R
```

where R, G, and B are the values of the R, G, and B components. The single computed L component replaces the R, G, and B components in the group.

(modify the section labeled "Final Conversion", p. 222)

For a floating-point RGBA color, each component is first clamped to [0, 1]. Then the appropriate conversion formula from table 4.7 is applied to the component. For an integer RGBA color, each component is clamped to the representable range of <type>.

Additions to Chapter 6 of the OpenGL 2.0 Specification (State and State Requests)

Modify Section 6.1.3 (Enumerated Queries), p. 246

(insert in the list of query functions, p. 246)

```
void GetTexParameterIivEXT( enum target, enum value, int *data );
void GetTexParameterIuivEXT( enum target, enum value, uint *data );
```

(modify the second paragraph, p. 247)
... For GetTexParameter, value must be either TEXTURE_RESIDENT, or one of the symbolic values in table 3.19. Querying <value> TEXTURE_BORDER_COLOR with GetTexParameterIivEXT or GetTexParameterIuivEXT returns the border color values as signed integers or unsigned integers, respectively; otherwise the values are returned as described in section 6.1.2. If the border color is queried with a type that does not match the original type with which it was specified, the result is undefined. The <lod> argument ...

(add to end of third paragraph, p. 247) Queries with a <value> of TEXTURE_RED_TYPE_ARB, TEXTURE_GREEN_TYPE_ARB, TEXTURE_BLUE_TYPE_ARB, TEXTURE_ALPHA_TYPE_ARB, TEXTURE_LUMINANCE_TYPE_ARB, TEXTURE_INTENSITY_TYPE_ARB, or TEXTURE_DEPTH_TYPE_ARB, return the data type used to store the component. Values of NONE, UNSIGNED_NORMALIZED_ARB, FLOAT, INT, or UNSIGNED_INT, indicate missing,

    unsigned normalized integer, floating-point, signed unnormalized integer,
    and unsigned unnormalized integer components, respectively.

GLX Protocol

    TBD

Dependencies on ARB_texture_float

    The following changes should be made if ARB_texture_float is not
    supported:

    The references to floating-point data types in section 3.8, p. 150
    should be deleted.

    The language in section 3.8.1 should indicate that final
    conversion always clamps when the internalformat is not integer.

    The description of table 3.16 should not mention the <f>
    floating-point formats.

    Section 3.8.4 should indicate that border color values should be
    clamped to [0,1] before being stored, if not specified with one of
    the TexParameterI* functions.

    Section 3.8.8 should not mention clamping border color values to
    [0,1] for fixed-point textures, since this occurs in 3.8.4 at
    TexParameter specification.

Dependencies on ARB_color_buffer_float

    The following changes should be made if ARB_color_buffer_float is
    not supported:

    Section 3.11.2, subsection "Shader Outputs: p. 196 should not
    mention fragment clamping or color buffers with floating-point
    formats.

    Chapter 4, p. 198 should not mention components represented as
    floating-point values.

    Section 4.1.3, p. 200, section 4.1.4 p. 205, section 4.1.8 p. 206,
    section 4.2.3 p. 215 and section 4.3.2 p. 222 should not mention
    color buffers with a floating-point format.

    Section 4.2.3 p. 217 should not mention clamping the clear color
    values to [0,1].

Errors

    INVALID_OPERATION is generated by Begin, DrawPixels, Bitmap,
    CopyPixels, or a command that performs an explicit Begin if the
    color buffer has an integer RGBA format and no fragment shader is
    active.

    INVALID_ENUM is generated by DrawPixels, TexImage* and
    SubTexImage* if <format> is one of the integer component formats

```
    described in table 3.6 and <type> is FLOAT.

    INVALID_OPERATION is generated by TexImage* and SubTexImage* if
    the texture internalformat is an integer format as described in
    table 3.16 and <format> is not one of the integer component
    formats described in table 3.6, or if the internalformat is not an
    integer format and <format> is an integer format.

    INVALID_OPERATION is generated by CopyTexImage* and
    CopyTexSubImage* if the texture internalformat is an integer
    format and the read color buffer is not an integer format, or if
    the internalformat is not an integer format and the read color
    buffer is an integer format.

    INVALID_ENUM is generated by ReadPixels if <format> is an integer
    format and <type> is FLOAT.

    INVALID_OPERATON is generated by ReadPixels if <format> is an
    integer format and the color buffer is not an integer format, or
    if <format> is not an integer format and the color buffer is an
    integer format.
```

New State

    (modify table 6.33, p. 294)

Get Value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
RGBA_INTEGER_MODE_EXT	B	GetBooleanv	-	True if RGBA components are integers	2.7	-

Issues

How should the integer pixel path be triggered: by the destination type, new source types, or new source formats?

    RESOLVED: New source formats, based on the precedence of
    COLOR_INDEX and STENCIL_INDEX formats which invoke distinct
    pixel path behavior with identical data types and independent
    of the destination.

Should pixel transfer operations be defined for the integer pixel path?

    RESOLVED: No. Fragment shaders can achieve similar results
    with more flexibility. There is no need to aggrandize this
    legacy mechanism.

What happens if a shader reads a float texel from an integer texture or vice-versa?

    RESOLVED: The result is undefined. The shader must have
    knowledge of the texture internal data type.

How do integer textures behave in fixed function fragment processing?

RESOLVED: The fixed function texture pipeline assumes textures return floating-point values, hence the return value from an integer texture will not be in a meaningful format.

How does TEXTURE_BORDER_COLOR work with integer textures?

RESOLVED: The internal storage of border values effectively becomes a union, and the returned values are interpreted as the same type as the texture. New versions of TexParameter allow specification of signed and unsigned integer border values.

How does logic op behave with RGBA mode rendering into integer color buffer?

RESOLVED: The color logic op operates when enabled when rendering into integer color buffers.

Logic op operations make sense for integer color buffers so the COLOR_LOGIC_OP enable is respected when rendering into integer color buffers.

Blending does not apply to RGBA mode rendering when rendering into integer color buffers (as section 4.1.8 is updated to say). The color logic op (described in section 4.1.10) is not a blending operation (though it does take priority over the blending enable).

Revision History

```
    Rev.    Date    Author    Changes
    ----  --------  --------  -----------------------------------------
      5   07/15/07  pbrown    Fix typo in GetTexParameterIuivEXT function
                              name in "New Procedures and Functions".

      4      --               Pre-release revisions.
```

Name

EXT_texture_lod_bias

Name Strings

GL_EXT_texture_lod_bias

Notice

Status

Shipping since late 1999.

The texture LOD bias functionality in OpenGL 1.4 is based on this extension though the OpenGL 1.4 functionality added the ability to specify a second per-texture object bias term. The OpenGL 1.4 enum values match the EXT enum values.

Version

NVIDIA Date: August 27, 2003
$Date: 2003/08/27 $ $Revision: #13 $

Number

186

Dependencies

Written based on the wording of the OpenGL 1.2 specification.

Affects ARB_multitexture.

Overview

OpenGL computes a texture level-of-detail parameter, called lambda in the GL specification, that determines which mipmap levels and their relative mipmap weights for use in mipmapped texture filtering.

This extension provides a means to bias the lambda computation by a constant (signed) value. This bias can provide a way to blur or pseudo-sharpen OpenGL's standard texture filtering.

This blurring or pseudo-sharpening may be useful for special effects (such as depth-of-field effects) or image processing techniques (where the mipmap levels act as pre-downsampled image versions). On some implementations, increasing the texture lod bias may improve texture filtering performance (at the cost of texture bluriness).

The extension mimics functionality found in Direct3D.

Issues

Should the texture LOD bias be settable per-texture object or *per-texture stage?*

RESOLUTION: Per-texture stage. This matches the Direct3D semantics for texture lod bias. Note that this differs from the semantics of SGI's SGIX_texture_lod_bias extension that has the biases per-texture object.

This also allows the same texture object to be used by two different texture units for different blurring. This is useful for extrapolating detail between various levels of detail in a mipmapped texture.

For example, you can extrapolate texture detail with ARB_multitexture and EXT_texture_env_combine by computing

```
  (B0 - B2) * 2 + B2
```

where B0 is a non-biased texture (normal sharpness) and B2 is the same texture but bias by 2 levels-of-detail (fairly blurry). This has the effect of increasing the high-frequency information in the texture. There are immediate Earth Sciences and medical imaging applications for this technique.

Per-texture stage control of the LOD bias is also useful for allowing an application to control overall texture bluriness. This can be used in games to simulate disorientation (note that only textures will blur, not edges). It can also be used to globally control texturing performance. An application may be able to sustain a constant frame rate by avoiding texture fetch stalls by using slightly blurrier textures.

How does EXT_texture_lod_bias differ from SGIX_texture_lod bias?

EXT_texture_lod_bias adds a bias to lambda. The SGIX_texture_lod_bias extension changes the computation of rho (the log2 of which is lambda). The SGIX extension provides separate biases in each texture dimension. The EXT extension does not provide an "directionality" in the LOD control.

Does the texture lod bias occur before or after the TEXTURE_MAX_LOD and TEXTURE_MIN_LOD clamping?

RESOLUTION: BEFORE. This allows the texture lod bias to still be clamped within the max/min lod range.

Does anything special have to be said to keep the biased lambda value from being less than zero or greater than the maximum number of mipmap levels?

RESOLUTION: NO. The existing clamping in the specification handles these situations.

The texture lod bias is specified to be a float. In practice, what sort of range is assumed for the texture lod bias?

RESOLUTION: The MAX_TEXTURE_LOD_BIAS_EXT implementation constant advertises the maximum absolute value of the supported texture lod bias. The value is recommended to be at least the maximum mipmap level supported by the implementation.

The texture lod bias is specified to be a float. In practice, what sort of precision is assumed for the texture lod bias?

RESOLUTION; This is implementation dependent. Presumably, hardware would implement the texture lod bias as a fractional bias but the exact fractional precision supported is implementation dependent. At least 4 fractional bits is recommended.

New Procedures and Functions

None

New Tokens

Accepted by the <target> parameters of GetTexEnvfv, GetTexEnviv, TexEnvi, TexEnvf, Texenviv, and TexEnvfv:

```
    TEXTURE_FILTER_CONTROL_EXT          0x8500
```

When the <target> parameter of GetTexEnvfv, GetTexEnviv, TexEnvi, TexEnvf, TexEnviv, and TexEnvfv is TEXTURE_FILTER_CONTROL_EXT, then the value of <pname> may be:

```
    TEXTURE_LOD_BIAS_EXT                0x8501
```

Accepted by the <pname> parameters of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

```
    MAX_TEXTURE_LOD_BIAS_EXT           0x84FD
```

Additions to Chapter 2 of the 1.2 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.2 Specification (Rasterization)

-- Section 3.8.5 "Texture Minification"

Change the first formula under "Scale Factor and Level of Detail" to read:

"The choice is governed by a scale factor p(x,y), the level of detail parameter lambda(x,y), defined as

```
            lambda'(x,y) = log2[p(x,y)] + lodBias
```

where lodBias is the texture unit's (signed) texture lod bias parameter (as described in Section 3.8.9) clamped between the positive and negative values of the implementation defined constant MAX_TEXTURE_LOD_BIAS_EXT."

-- Section 3.8.9 "Texture Environments and Texture Functions"

Change the first paragraph to read:

"The command

```
   void TexEnv{if}(enum target, enum pname, T param);
   void TexEnv{if}v(enum target, enum pname, T params);
```

sets parameters of the texture environment that specifies how texture values are interepreted when texturing a fragment or sets per-texture unit texture filtering parameters. The possible target parameters are TEXTURE_ENV or TEXTURE_FILTER_CONTROL_EXT. ... When target is TEXTURE_ENV, the possible environment parameters are TEXTURE_ENV_MODE and TEXTURE_ENV_COLOR. ... When target is TEXTURE_FILTER_CONTROL_EXT, the only possible texture filter parameter is TEXTURE_LOD_BIAS_EXT. TEXTURE_LOD_BIAS_EXT is set to a signed floating point value that is used to bias the level of detail parameter, lambda, as described in Section 3.8.5."

Add a final paragraph at the end of the section:

"The state required for the per-texture unit filtering parameters consists of one floating-point value."

Additions to Chapter 4 of the 1.2 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.2 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

-- Section 6.1.3 "Texture Environments and Texture Functions"

Change the third sentence of the third paragraph to read:

"The env argument to GetTexEnv must be either TEXTURE_ENV or TEXTURE_FILTER_CONTROL_EXT."

Additions to the GLX Specification

None

Errors

INVALID_ENUM is generated when TexEnv is called with a <pname> of TEXTURE_FILTER_CONTROL_EXT and the value of <param> or what is pointed to by <params> is not TEXTURE_LOD_BIAS_EXT.

New State

(table 6.14, p204) add the entry:

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
TEXTURE_LOD_BIAS_EXT	R	GetTexEnvfv	0.0	Biases texture level of detail	3.8.9	texture

(When ARB_multitexture is supported, the TEXTURE_LOD_BIAS_EXT state is per-texture unit.)

New Implementation State

(table 6.24, p214) add the following entries:

Get Value	Type	Get Command	Minimum Value	Description	Sec	Attribute
MAX_TEXTURE_LOD_BIAS_EXT	R+	GetFloatv	4.0	Maximum absolute texture lod bias	3.8.9	-

Revision History

8/27/03 - updated status to mention OpenGL 1.4 functionality

8/26/03 - fixed incorrect enum name (TEXTURE_FILTER_CONTROL_EXT is correct) in the Errors section.

6/2/00 - add spec language to allow GetTexEnv to accept TEXTURE_FILTER_CONTROL_EXT.

Name

```
    EXT_texture_mirror_clamp
```

Name Strings

```
    GL_EXT_texture_mirror_clamp
```

Status

```
    Shipping as of May 2004 for GeForce6.
```

Version

```
    Last Modified Date:  $Date: 2004/05/17 $
    NVIDIA Revision: $Revision: #4 $
```

Number

```
    298
```

Issues

How does EXT_texture_mirror_clamp extend ATI_texture_mirror_once?

```
        This EXT extension provides the two wrap modes that
        ATI_texture_mirror_once adds but also adds a third new wrap mode
        (GL_MIRROR_CLAMP_TO_BORDER_EXT).  This extension uses the same
        enumerant values for the ATI_texture_mirror_once modes.
```

Why is the GL_MIRROR_CLAMP_TO_BORDER_EXT mode more interesting than the two other modes?

```
        Rather than clamp to 100% of the edge of the texture
        (GL_MIRROR_CLAMP_TO_EDGE_EXT) or to 50% of the edge and border
        color (GL_MIRROR_CLAMP), it is preferable to clamp to 100%
        of the border color (GL_MIRROR_CLAMP_TO_BORDER_EXT).  This
        avoids "bleeding" at smaller mipmap levels.

        Consider a texture that encodes a circular fall-off pattern such
        as for a projected spotlight.  A circular pattern is bi-symmetric
        so a "mirror clamp" wrap modes can reduce the memory footprint
        of the texture by a fourth.  Far outside the spotlight pattern,
        you'd like to sample 100% of the border color (typically black
        for a spotlight texture).  The way to achieve this without any
        bleeding of edge texels is with GL_MIRROR_CLAMP_TO_BORDER_EXT.
```

Does this extension complete the orthogonality of the current five OpenGL 1.5 wrap modes?

Yes. There are two ways for repetition to operate (repeated & mirrored) and four ways for texture coordinate clamping to operate (unclamped, clamp, clamp to edge, & clamp to border). The complete table of all 8 modes looks like this:

	Repeat	Mirror
Unclamped	REPEAT	MIRRORED_REPEAT
Clamp	CLAMP	MIRROR_CLAMP
Clamp to edge	CLAMP_TO_EDGE	MIRROR_CLAMP_TO_EDGE
Clamp to border	CLAMP_TO_BORDER	MIRROR_CLAMP_TO_BORDER

OpenGL 1.0 introduced REPEAT & CLAMP.
OpenGL 1.2 introduced CLAMP_TO_EDGE
OpenGL 1.3 introduced CLAMP_TO_BORDER
OpenGL 1.4 introduced MIRRORED_REPEAT
ATI_texture_mirror_once introduced MIRROR_CLAMP & MIRROR_CLAMP_TO_EDGE
EXT_texture_mirror_clamp introduces MIRROR_CLAMP_TO_BORDER

Do these three new wrap modes work with 1D, 2D, 3D, and cube map texture targets?

RESOLUTION: Yes.

Do these three new wrap modes work with ARB_texture_non_power_of_two functionality?

RESOLUTION: Yes.

Do these three new wrap modes interact with NV_texture_rectangle?

RESOLUTION: Mirroring wrap modes are not supported by GL_TEXTURE_RECTANGLE_NV textures. Conventional mirroring is already not supported for texture rectangles so supporting clamped mirroring modes should not be supported either.

Does the specification of MIRROR_CLAMP_EXT & MIRROR_CLAMP_TO_EDGE_EXT match the ATI_texture_mirror_once specification?

I believe yes. The ATI_texture_mirror_once specification is somewhat vague what happens to texture coordinates at or very near (within half a texel of) zero. The presumption is that a CLAMP_TO_EDGE behavior is used. This specification is quite explicit that values near zero are clamped to plus or minus 1/(2*N) respectively so that the CLAMP_TO_EDGE behavior is explicit.

What should this extension be called?

Calling the extension EXT_texture_mirror_once might cause confusion since this extension has additional functionality. Also, "once" never appears in the specification. EXT_texture_mirror_clamp is a good name because it implies support for all the clamped versions of mirroring.

There is GL_MIRRORED_REPEAT and then GL_MIRROR_CLAMP_EXT, GL_MIRROR_CLAMP_TO_EDGE_EXT, and GL_MIRROR_CLAMP_TO_BORDER_EXT. Why does the first enumerant name say "MIRRORED" while the other three say "MIRROR"?

This extension follows the naming precedent set by the ATI_texture_mirror_once specification.

Moreover, MIRRORED_REPEAT uses "mirrored" to help that the mirroring repeats infinitely. For the other three modes, there is just one mirror that occurs and then a clamp.

Dependencies

Written based on the wording of the OpenGL 1.4.

Extends ATI_texture_mirror_once by adding GL_MIRROR_CLAMP_TO_BORDER_EXT.

NV_texture_rectangle trivially affects the definition of this extension.

Overview

EXT_texture_mirror_clamp extends the set of texture wrap modes to include three modes (GL_MIRROR_CLAMP_EXT, GL_MIRROR_CLAMP_TO_EDGE_EXT, GL_MIRROR_CLAMP_TO_BORDER_EXT) that effectively use a texture map twice as large as the original image in which the additional half of the new image is a mirror image of the original image.

This new mode relaxes the need to generate images whose opposite edges match by using the original image to generate a matching "mirror image". This mode allows the texture to be mirrored only once in the negative s, t, and r directions.

New Procedure and Functions

None

New Tokens

Accepted by the <param> parameter of TexParameteri and TexParameterf, and by the <params> parameter of TexParameteriv and TexParameterfv, when their <pname> parameter is TEXTURE_WRAP_S, TEXTURE_WRAP_T, or TEXTURE_WRAP_R:

```
  MIRROR_CLAMP_EXT                  0x8742 (same value as MIRROR_CLAMP_ATI)
  MIRROR_CLAMP_TO_EDGE_EXT          0x8743 (same value as MIRROR_CLAMP_TO_EDGE_ATI)
  MIRROR_CLAMP_TO_BORDER_EXT        0x8912
```

Additions to Chapter 2 of the OpenGL 1.2.1 Specification (Operation)

None

Additions to Chapter 3 if the OpenGL 1.2.1 Specification (Rasterization):

- (3.8.4, page 136, as amended by the NV_texture_rectangle extension)

  Add the 3 new wrap modes to the list of wrap modes unsupported for the TEXTURE_RECTANGLE_NV texture target.

  "Certain texture parameter values may not be specified for textures with a target of TEXTURE_RECTANGLE_NV. The error INVALID_ENUM is generated if the target is TEXTURE_RECTANGLE_NV and the TEXTURE_WRAP_S, TEXTURE_WRAP_T, or TEXTURE_WRAP_R parameter is set to REPEAT, MIRRORED_REPEAT_IBM, MIRROR_CLAMP_EXT, MIRROR_CLAMP_TO_EDGE_EXT, and MIRROR_CLAMP_TO_BORDER_EXT. The error INVALID_ENUM is generated if the target is TEXTURE_RECTANGLE_NV and the TEXTURE_MIN_FILTER is set to a value other than NEAREST or LINEAR (no mipmap filtering is permitted). The error INVALID_ENUM is generated if the target is TEXTURE_RECTANGLE_NV and TEXTURE_BASE_LEVEL is set to any value other than zero."

- Table 3.19, page 137: Change first three entries in table:

"TEXTURE_WRAP_S	integer	CLAMP, CLAMP_TO_BORDER, CLAMP_TO_EDGE, MIRRORED_REPEAT, MIRROR_CLAMP_EXT, MIRROR_CLAMP_TO_BORDER_EXT, MIRROR_CLAMP_TO_EDGE_EXT, REPEAT
TEXTURE_WRAP_T	integer	CLAMP, CLAMP_TO_BORDER, CLAMP_TO_EDGE, MIRRORED_REPEAT, MIRROR_CLAMP_EXT, MIRROR_CLAMP_TO_BORDER_EXT, MIRROR_CLAMP_TO_EDGE_EXT, REPEAT
TEXTURE_WRAP_R	integer	CLAMP, CLAMP_TO_BORDER, CLAMP_TO_EDGE, MIRRORED_REPEAT, MIRROR_CLAMP_EXT, MIRROR_CLAMP_TO_BORDER_EXT, MIRROR_CLAMP_TO_EDGE_EXT, REPEAT"

- (3.8.7, page 140) After the last paragraph of the section add:

  "Wrap Mode MIRROR_CLAMP_EXT

  Wrap mode MIRROR_CLAMP_EXT mirrors and clamps the texture coordinate, where mirroring and clamping a value f computes

```
  mirrorClamp(f) = min(1, max(1/(2*N), abs(f)))
```

  where N is the size of the one-, two-, or three-dimensional texture image in the direction of wrapping.

  Wrap Mode MIRROR_CLAMP_TO_EDGE_EXT

  Wrap mode MIRROR_CLAMP_TO_EDGE_EXT mirrors and clamps to edge the texture coordinate, where mirroring and clamping to edge a value f computes

```
  mirrorClampToEdge(f) = min(1-1/(2*N), max(1/(2*N), abs(f)))
```

  where N is the size of the one-, two-, or three-dimensional texture image in the direction of wrapping.

Wrap Mode MIRROR_CLAMP_TO_BORDER_EXT

Wrap mode MIRROR_CLAMP_TO_BORDER_EXT mirrors and clamps to border the texture coordinate, where mirroring and clamping to border a value f computes

```
  mirrorClampToBorder(f) = min(1+1/(2*N), max(1/(2*N), abs(f)))
```

where N is the size of the one-, two-, or three-dimensional texture image in the direction of wrapping."

- (3.8.8, page 142) Delete this phrase because it is out of date and unnecessary given the current way section 3.8.7 is written:

  "(if the wrap mode for a coordinate is CLAMP or CLAMP_TO_EDGE)"

Additions to Chapter 4:

None

Additions to Chapter 5:

None

Additions to Chapter 6:

None

Additions to the GLX Specification

None

Dependencies on NV_texture_rectangle

If NV_texture_rectangle is not supported, ignore the statement that the initial value for the S, T, and R wrap modes is CLAMP_TO_EDGE for rectangular textures.

Ignore the error for a texture target of TEXTURE_RECTANGLE_NV.

GLX Protocol

None

Errors

INVALID_ENUM is generated when TexParameter is called with a target of TEXTURE_RECTANGLE_NV and the TEXTURE_WRAP_S, TEXTURE_WRAP_T, or TEXTURE_WRAP_R parameter is set to REPEAT, MIRRORED_REPEAT_IBM, MIRROR_CLAMP_EXT, MIRROR_CLAMP_TO_EDGE_EXT, or MIRROR_CLAMP_TO_BORDER_EXT.

New State

(table 6.15, p230) amend the following entries [Z5 changed to Z8]:

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
TEXTURE_WRAP_S	n*Z8	GetTexParameter	REPEAT except for rectangular which is CLAMP_TO_EDGE	Texture wrap mode S	3.8.7	texture
TEXTURE_WRAP_T	n*Z8	GetTexParameter	REPEAT except for rectangular which is CLAMP_TO_EDGE	Texture wrap mode T	3.8.7	texture
TEXTURE_WRAP_R	n*Z8	GetTexParameter	REPEAT except for rectangular which is CLAMP_TO_EDGE	Texture wrap mode R	3.8.7	texture

New Implementation Dependent State

None

Name

    EXT_texture_object

Name Strings

    GL_EXT_texture_object

Version

    $Date: 1995/10/03 05:39:56 $ $Revision: 1.27 $

Number

    20

Dependencies

    EXT_texture3D affects the definition of this extension

Overview

    This extension introduces named texture objects. The only way to name
    a texture in GL 1.0 is by defining it as a single display list. Because
    display lists cannot be edited, these objects are static. Yet it is
    important to be able to change the images and parameters of a texture.

Issues

    *   Should the dimensions of a texture object be static once they are
        changed from zero? This might simplify the management of texture
        memory. What about other properties of a texture object?

        No.

Reasoning

    *   Previous proposals overloaded the <target> parameter of many Tex
        commands with texture object names, as well as the original
        enumerated values. This proposal eliminated such overloading,
        choosing instead to require an application to bind a texture object,
        and then operate on it through the binding reference. If this
        constraint ultimately proves to be unacceptable, we can always
        extend the extension with additional binding points for editing and
        querying only, but if we expect to do this, we might choose to bite
        the bullet and overload the <target> parameters now.

    *   Commands to directly set the priority of a texture object and to
        query the resident status of a texture object are included. I feel
        that binding a texture object would be an unacceptable burden for
        these management operations. These commands also allow queries and
        operations on lists of texture objects, which should improve
        efficiency.

    *   GenTexturesEXT does not return a success/failure boolean because
        it should never fail in practice.

New Procedures and Functions

```
    void GenTexturesEXT(sizei n,
                        uint* textures);

    void DeleteTexturesEXT(sizei n,
                           const uint* textures);

    void BindTextureEXT(enum target,
                        uint texture);

    void PrioritizeTexturesEXT(sizei n,
                               const uint* textures,
                               const clampf* priorities);

    boolean AreTexturesResidentEXT(sizei n,
                                   const uint* textures,
                                   boolean* residences);

    boolean IsTextureEXT(uint texture);
```

New Tokens

```
    Accepted by the <pname> parameters of TexParameteri, TexParameterf,
    TexParameteriv, TexParameterfv, GetTexParameteriv, and GetTexParameterfv:

        TEXTURE_PRIORITY_EXT            0x8066

    Accepted by the <pname> parameters of GetTexParameteriv and
    GetTexParameterfv:

        TEXTURE_RESIDENT_EXT            0x8067

    Accepted by the <pname> parameters of GetBooleanv, GetIntegerv,
    GetFloatv, and GetDoublev:

        TEXTURE_1D_BINDING_EXT          0x8068
        TEXTURE_2D_BINDING_EXT          0x8069
        TEXTURE_3D_BINDING_EXT          0x806A
```

Additions to Chapter 2 of the 1.0 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.0 Specification (Rasterization)

Add the following discussion to section 3.8 (Texturing). In addition to the default textures TEXTURE_1D, TEXTURE_2D, and TEXTURE_3D_EXT, it is possible to create named 1, 2, and 3-dimensional texture objects. The name space for texture objects is the unsigned integers, with zero reserved by the GL.

A texture object is created by binding an unused name to TEXTURE_1D, TEXTURE_2D, or TEXTURE_3D_EXT. This binding is accomplished by calling BindTextureEXT with <target> set to TEXTURE_1D, TEXTURE_2D, or TEXTURE_3D_EXT, and <texture> set to the name of the new texture object.

When a texture object is bound to a target, the previous binding for that target is automatically broken.

When a texture object is first bound it takes the dimensionality of its target. Thus, a texture object first bound to TEXTURE_1D is 1-dimensional; a texture object first bound to TEXTURE_2D is 2-dimensional, and a texture object first bound to TEXTURE_3D_EXT is 3-dimensional. The state of a 1-dimensional texture object immediately after it is first bound is equivalent to the state of the default TEXTURE_1D at GL initialization. Likewise, the state of a 2-dimensional or 3-dimensional texture object immediately after it is first bound is equivalent to the state of the default TEXTURE_2D or TEXTURE_3D_EXT at GL initialization. Subsequent bindings of a texture object have no effect on its state. The error INVALID_OPERATION is generated if an attempt is made to bind a texture object to a target of different dimensionality.

While a texture object is bound, GL operations on the target to which it is bound affect the bound texture object, and queries of the target to which it is bound return state from the bound texture object. If texture mapping of the dimensionality of the target to which a texture object is bound is active, the bound texture object is used.

By default when an OpenGL context is created, TEXTURE_1D, TEXTURE_2D, and TEXTURE_3D_EXT have 1, 2, and 3-dimensional textures associated with them. In order that access to these default textures not be lost, this extension treats them as though their names were all zero. Thus the default 1-dimensional texture is operated on, queried, and applied as TEXTURE_1D while zero is bound to TEXTURE_1D. Likewise, the default 2-dimensional texture is operated on, queried, and applied as TEXTURE_2D while zero is bound to TEXTURE_2D, and the default 3-dimensional texture is operated on, queried, and applied as TEXTURE_3D_EXT while zero is bound to TEXTURE_3D_EXT.

Texture objects are deleted by calling DeleteTexturesEXT with <textures> pointing to a list of <n> names of texture object to be deleted. After a texture object is deleted, it has no contents or dimensionality, and its name is freed. If a texture object that is currently bound is deleted, the binding reverts to zero. DeleteTexturesEXT ignores names that do not correspond to textures objects, including zero.

GenTexturesEXT returns <n> texture object names in <textures>. These names are chosen in an unspecified manner, the only condition being that only names that were not in use immediately prior to the call to GenTexturesEXT are considered. Names returned by GenTexturesEXT are marked as used (so that they are not returned by subsequent calls to GenTexturesEXT), but they are associated with a texture object only after they are first bound (just as if the name were unused).

An implementation may choose to establish a working set of texture objects on which binding operations are performed with higher performance. A texture object that is currently being treated as a part of the working set is said to be resident. AreTexturesResidentEXT returns TRUE if all of the <n> texture objects named in <textures> are resident, FALSE otherwise. If FALSE is returned, the residence of each texture object is returned in <residences>. Otherwise the contents of the <residences> array are not changed. If any of the names in

<textures> is not the name of a texture object, FALSE is returned, the error INVALID_VALUE is generated, and the contents of <residences> are indeterminate. The resident status of a single bound texture object can also be queried by calling GetTexParameteriv or GetTexParameterfv with <target> set to the target to which the texture object is bound, and <pname> set to TEXTURE_RESIDENT_EXT. This is the only way that the resident status of a default texture can be queried.

Applications guide the OpenGL implementation in determining which texture objects should be resident by specifying a priority for each texture object. PrioritizeTexturesEXT sets the priorities of the <n> texture objects in <textures> to the values in <priorities>. Each priority value is clamped to the range [0.0, 1.0] before it is assigned. Zero indicates the lowest priority, and hence the least likelihood of being resident. One indicates the highest priority, and hence the greatest likelihood of being resident. The priority of a single bound texture object can also be changed by calling TexParameteri, TexParameterf, TexParameteriv, or TexParameterfv with <target> set to the target to which the texture object is bound, <pname> set to TEXTURE_PRIORITY_EXT, and <param> or <params> specifying the new priority value (which is clamped to [0.0,1.0] before being assigned). This is the only way that the priority of a default texture can be specified. (PrioritizeTexturesEXT silently ignores attempts to prioritize nontextures, and texture zero.)

Additions to Chapter 4 of the 1.0 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.0 Specification (Special Functions)

BindTextureEXT and PrioritizeTexturesEXT are included in display lists. All other commands defined by this extension are not included in display lists.

Additions to Chapter 6 of the 1.0 Specification (State and State Requests)

IsTextureEXT returns TRUE if <texture> is the name of a valid texture object. If <texture> is zero, or is a non-zero value that is not the name of a texture object, or if an error condition occurs, IsTextureEXT returns FALSE.

Because the query values of TEXTURE_1D, TEXTURE_2D, and TEXTURE_3D_EXT are already defined as booleans indicating whether these textures are enabled or disabled, another mechanism is required to query the binding associated with each of these texture targets. The name of the texture object currently bound to TEXTURE_1D is returned in <params> when GetIntegerv is called with <pname> set to TEXTURE_1D_BINDING_EXT. If no texture object is currently bound to TEXTURE_1D, zero is returned. Likewise, the name of the texture object bound to TEXTURE_2D or TEXTURE_3D_EXT is returned in <params> when GetIntegerv is called with <pname> set to TEXTURE_2D_BINDING_EXT or TEXTURE_3D_BINDING_EXT. If no texture object is currently bound to TEXTURE_2D or to TEXTURE_3D_EXT, zero is returned.

A texture object comprises the image arrays, priority, border color, filter modes, and wrap modes that are associated with that object. More explicitly, the state list

```
    TEXTURE,
    TEXTURE_PRIORITY_EXT
    TEXTURE_RED_SIZE,
    TEXTURE_GREEN_SIZE,
    TEXTURE_BLUE_SIZE,
    TEXTURE_ALPHA_SIZE,
    TEXTURE_LUMINANCE_SIZE,
    TEXTURE_INTENSITY_SIZE,
    TEXTURE_WIDTH,
    TEXTURE_HEIGHT,
    TEXTURE_DEPTH_EXT,
    TEXTURE_BORDER,
    TEXTURE_COMPONENTS,
    TEXTURE_BORDER_COLOR,
    TEXTURE_MIN_FILTER,
    TEXTURE_MAG_FILTER,
    TEXTURE_WRAP_S,
    TEXTURE_WRAP_T,
    TEXTURE_WRAP_R_EXT
```

composes a single texture object.

When PushAttrib is called with TEXTURE_BIT enabled, the priorities, border colors, filter modes, and wrap modes of the currently bound texture objects are pushed, as well as the current texture bindings and enables. When an attribute set that includes texture information is popped, the bindings and enables are first restored to their pushed values, then the bound texture objects have their priorities, border colors, filter modes, and wrap modes restored to their pushed values.

Additions to the GLX Specification

Texture objects are shared between GLX rendering contexts if and only if the rendering contexts share display lists. No change is made to the GLX API.

GLX Protocol

Six new GL commands are added.

The following rendering command is sent to the server as part of a glXRender request:

```
    BindTextureEXT
        2           12              rendering command length
        2           4117            rendering command opcode
        4           ENUM            target
        4           CARD32          texture
```

The following rendering command can be sent to the server as part of a glXRender request or as part of a glXRenderLarge request:

```
    PrioritizeTexturesEXT
        2           8+(n*8)         rendering command length
        2           4118            rendering command opcode
        4           INT32           n
        n*4         LISTofCARD32    textures
        n*4         LISTofFLOAT32   priorities

        If the command is encoded in a glXRenderLarge request, the
        command opcode and command length fields above are expanded to
        4 bytes each:

        4           12+(n*8)        rendering command length
        4           4118            rendering command opcode
```

```
The remaining commands are non-rendering commands. These commands are
sent separately (i.e., not as part of a glXRender or glXRenderLarge
request), using either the glXVendorPrivate request or the
glXVendorPrivateWithReply request:

    DeleteTexturesEXT
        1           CARD8           opcode (X assigned)
        1           16              GLX opcode (glXVendorPrivate)
        2           4+n             request length
        4           12              vendor specific opcode
        4           GLX_CONTEXT_TAG context tag
        4           INT32           n
        n*4         CARD32          textures

    GenTexturesEXT
        1           CARD8           opcode (X assigned)
        1           17              GLX opcode (glXVendorPrivateWithReply)
        2           4               request length
        4           13              vendor specific opcode
        4           GLX_CONTEXT_TAG context tag
        4           INT32           n
      =>
        1           1               reply
        1                           unused
        2           CARD16          sequence number
        4           n               reply length
        24                          unused
        4*n         LISTofCARD32    textures
    AreTexturesResidentEXT
        1           CARD8           opcode (X assigned)
        1           17              GLX opcode (glXVendorPrivateWithReply)
        2           4+n             request length
        4           11              vendor specific opcode
        4           GLX_CONTEXT_TAG context tag
        4           INT32           n
        4*n         LISTofCARD32    textures
      =>
        1           1               reply
        1                           unused
        2           CARD16          sequence number
        4           (n+p)/4         reply length
        4           BOOL32          return_value
        20                          unused
        n           LISTofBOOL      residences
        p                           unused, p=pad(n)

    IsTextureEXT
        1           CARD8           opcode (X assigned)
        1           17              GLX opcode (glXVendorPrivateWithReply)
        2           4               request length
        4           14              vendor specific opcode
        4           GLX_CONTEXT_TAG context tag
        4           CARD32          textures
      =>
        1           1               reply
        1                           unused
        2           CARD16          sequence number
        4           0               reply length
        4           BOOL32          return_value
        20                          unused
```

Dependencies on EXT_texture3D

If EXT_texture3D is not supported, then all references to 3D textures in this specification are invalid.

Errors

INVALID_VALUE is generated if GenTexturesEXT parameter <n> is negative.

INVALID_VALUE is generated if DeleteTexturesEXT parameter <n> is negative.

INVALID_ENUM is generated if BindTextureEXT parameter <target> is not TEXTURE_1D, TEXTURE_2D, or TEXTURE_3D_EXT.

INVALID_OPERATION is generated if BindTextureEXT parameter <target> is TEXTURE_1D, and parameter <texture> is the name of a 2-dimensional or 3-dimensional texture object.

INVALID_OPERATION is generated if BindTextureEXT parameter <target> is TEXTURE_2D, and parameter <texture> is the name of a 1-dimensional or 3-dimensional texture object.

INVALID_OPERATION is generated if BindTextureEXT parameter <target> is

    TEXTURE_3D_EXT, and parameter <texture> is the name of a 1-dimensional
    or 2-dimensional texture object.

    INVALID_VALUE is generated if PrioritizeTexturesEXT parameter <n>
    negative.

    INVALID_VALUE is generated if AreTexturesResidentEXT parameter <n>
    is negative.

    INVALID_VALUE is generated by AreTexturesResidentEXT if any of the
    names in <textures> is zero, or is not the name of a texture.

    INVALID_OPERATION is generated if any of the commands defined in this
    extension is executed between the execution of Begin and the
    corresponding execution of End.

New State

Get Get Value	Get Command	Type	Initial Value	Attribute
TEXTURE_1D	IsEnabled	B	FALSE	texture/enable
TEXTURE_2D	IsEnabled	B	FALSE	texture/enable
TEXTURE_3D_EXT	IsEnabled	B	FALSE	texture/enable
TEXTURE_1D_BINDING_EXT	GetIntegerv	Z+	0	texture
TEXTURE_2D_BINDING_EXT	GetIntegerv	Z+	0	texture
TEXTURE_3D_BINDING_EXT	GetIntegerv	Z+	0	texture
TEXTURE_PRIORITY_EXT	GetTexParameterfv	n x Z+	1	texture
TEXTURE_RESIDENT_EXT	AreTexturesResidentEXT	n x B	unknown	-
TEXTURE	GetTexImage	n x levels x I	null	-
TEXTURE_RED_SIZE_EXT	GetTexLevelParameteriv	n x levels x Z+	0	-
TEXTURE_GREEN_SIZE_EXT	GetTexLevelParameteriv	n x levels x Z+	0	-
TEXTURE_BLUE_SIZE_EXT	GetTexLevelParameteriv	n x levels x Z+	0	-
TEXTURE_ALPHA_SIZE_EXT	GetTexLevelParameteriv	n x levels x Z+	0	-
TEXTURE_LUMINANCE_SIZE_EXT	GetTexLevelParameteriv	n x levels x Z+	0	-
TEXTURE_INTENSITY_SIZE_EXT	GetTexLevelParameteriv	n x levels x Z+	0	-
TEXTURE_WIDTH	GetTexLevelParameteriv	n x levels x Z+	0	-
TEXTURE_HEIGHT	GetTexLevelParameteriv	n x levels x Z+	0	-
TEXTURE_DEPTH_EXT	GetTexLevelParameteriv	n x levels x Z+	0	-
TEXTURE_4DSIZE_SGIS	GetTexLevelParameteriv	n x levels x Z+	0	-
TEXTURE_BORDER	GetTexLevelParameteriv	n x levels x Z+	0	-
TEXTURE_COMPONENTS (1D and 2D)	GetTexLevelParameteriv	n x levels x Z42	1	-
TEXTURE_COMPONENTS (3D and 4D)	GetTexLevelParameteriv	n x levels x Z38	LUMINANCE	-
TEXTURE_BORDER_COLOR	GetTexParameteriv	n x C	0, 0, 0, 0	texture
TEXTURE_MIN_FILTER	GetTexParameteriv	n x Z7	NEAREST_MIPMAP_LINEAR	texture
TEXTURE_MAG_FILTER	GetTexParameteriv	n x Z3	LINEAR	texture
TEXTURE_WRAP_S	GetTexParameteriv	n x Z2	REPEAT	texture
TEXTURE_WRAP_T	GetTexParameteriv	n x Z2	REPEAT	texture
TEXTURE_WRAP_R_EXT	GetTexParameteriv	n x Z2	REPEAT	texture
TEXTURE_WRAP_Q_SGIS	GetTexParameteriv	n x Z2	REPEAT	texture

New Implementation Dependent State

    None

Name

    EXT_texture_shared_exponent

Name Strings

    GL_EXT_texture_shared_exponent

Contact

    Mark J. Kilgard, NVIDIA Corporation (mjk 'at' nvidia.com)

Contributors

    Pat Brown
    Jon Leech

Status

    Shipping for GeForce 8 Series (November 2006)

Version

    Date: February 6, 2007
    Revision: 0.5

Number

    333

Dependencies

    OpenGL 1.1 required

    ARB_color_buffer_float affects this extension.

    EXT_framebuffer_object affects this extension.

    This extension is written against the OpenGL 2.0 (September 7,
    2004) specification.

Overview

    Existing texture formats provide either fixed-point formats with
    limited range and precision but with compact encodings (allowing 32
    or fewer bits per multi-component texel), or floating-point formats
    with tremendous range and precision but without compact encodings
    (typically 16 or 32 bits per component).

    This extension adds a new packed format and new internal texture
    format for encoding 3-component vectors (typically RGB colors) with
    a single 5-bit exponent (biased up by 15) and three 9-bit mantissas
    for each respective component. There is no sign bit so all three
    components must be non-negative. The fractional mantissas are
    stored without an implied 1 to the left of the decimal point.
    Neither infinity nor not-a-number (NaN) are representable in this
    shared exponent format.

This 32 bits/texel shared exponent format is particularly well-suited to high dynamic range (HDR) applications where light intensity is typically stored as non-negative red, green, and blue components with considerable range.

New Procedures and Functions

None

New Tokens

Accepted by the <internalformat> parameter of TexImage1D, TexImage2D, TexImage3D, CopyTexImage1D, CopyTexImage2D, and RenderbufferStorageEXT:

```
    RGB9_E5_EXT                                 0x8C3D
```

Accepted by the <type> parameter of DrawPixels, ReadPixels, TexImage1D, TexImage2D, GetTexImage, TexImage3D, TexSubImage1D, TexSubImage2D, TexSubImage3D, GetHistogram, GetMinmax, ConvolutionFilter1D, ConvolutionFilter2D, ConvolutionFilter3D, GetConvolutionFilter, SeparableFilter2D, GetSeparableFilter, ColorTable, ColorSubTable, and GetColorTable:

```
    UNSIGNED_INT_5_9_9_9_REV_EXT               0x8C3E
```

Accepted by the <pname> parameter of GetTexLevelParameterfv and GetTexLevelParameteriv:

```
    TEXTURE_SHARED_SIZE_EXT                     0x8C3F
```

Additions to Chapter 2 of the 2.0 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 2.0 Specification (Rasterization)

-- Section 3.6.4, Rasterization of Pixel Rectangles

Add a new row to Table 3.5 (page 128):

type Parameter Token Name	Corresponding GL Data Type	Special Interpretation
UNSIGNED_INT_5_9_9_9_REV_EXT	uint	yes

Add a new row to table 3.8: Packed pixel formats (page 132):

type Parameter Token Name	GL Data Type	Number of Components	Matching Pixel Formats
UNSIGNED_INT_5_9_9_9_REV_EXT	uint	4	RGB

Add a new entry to table 3.11: UNSIGNED_INT formats (page 134):

```
    UNSIGNED_INT_5_9_9_9_REV_EXT:

  31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
  +-------------+-------------------------+-------------------------+-------------------------+
  |  4th        |           3rd           |       2nd               |          1st            |
  +-------------+-------------------------+-------------------------+-------------------------+
```

Add to the end of the 2nd paragraph starting "Pixels are draw using":

"If type is UNSIGNED_INT_5_9_9_9_REV_EXT and format is not RGB then the error INVALID_ENUM occurs."

Add UNSIGNED_INT_5_9_9_9_REV_EXT to the list of packed formats in the 10th paragraph after the "Packing" subsection (page 130).

Add before the 3rd paragraph (page 135, starting "Calling DrawPixels with a type of BITMAP...") from the end of the "Packing" subsection:

"Calling DrawPixels with a type of UNSIGNED_INT_5_9_9_9_REV_EXT and format of RGB is a special case in which the data are a series of GL uint values. Each uint value specifies 4 packed components as shown in table 3.11. The 1st, 2nd, 3rd, and 4th components are called p_red, p_green, p_blue, and p_exp respectively and are treated as unsigned integers. These are then used to compute floating-point RGB components (ignoring the "Conversion to floating-point" section below in this case) as follows:

```
   red   = p_red   * 2^(p_exp - B)
   green = p_green * 2^(p_exp - B)
   blue  = p_blue  * 2^(p_exp - B)
```

where B is 15."

-- Section 3.8.1, Texture Image Specification:

"Alternatively if the internalformat is RGB9_E5_EXT, the red, green, and blue bits are converted to a shared exponent format according to the following procedure:

Components red, green, and blue are first clamped (in the process, mapping NaN to zero) so:

```
    red_c   = max(0, min(sharedexp_max, red))
    green_c = max(0, min(sharedexp_max, green))
    blue_c  = max(0, min(sharedexp_max, blue))
```

where sharedexp_max is (2^N-1)/2^N * 2^(Emax-B), N is the number of mantissa bits per component, Emax is the maximum allowed biased exponent value (careful: not necessarily 2^E-1 when E is the number of exponent bits), bits, and B is the exponent bias. For the RGB9_E5_EXT format, N=9, Emax=30 (careful: not 31!), and B=15.

The largest clamped component, max_c, is determined:

```
    max_c = max(red_c, green_c, blue_c)
```

A shared exponent is computed:

```
    exp_shared = max(-B-1, floor(log2(max_c))) + 1 + B
```

These integers values in the range 0 to 2^N-1 are then computed:

```
    red_s   = floor(red_c   / 2^(exp_shared - B + N) + 0.5)
    green_s = floor(green_c / 2^(exp_shared - B + N) + 0.5)
    blue_s  = floor(blue_c  / 2^(exp_shared - B + N) + 0.5)
```

Then red_s, green_s, and blue_s are stored along with exp_shared in the red, green, blue, and shared bits respectively of the texture image.

An implementation accepting pixel data of type UNSIGNED_INT_5_9_9_9_REV_EXT with a format of RGB is allowed to store the components "as is" if the implementation can determine the current pixel transfer state act as an identity transform on the components."

Add a new row and the "shared bits" column (blank for all existing rows) to Table 3.16 (page 154).

Sized Internal Format	Base Internal Format	R bits	G bits	B bits	A bits	L bits	I bits	D bits	shared bits
RGB9_E5_EXT	RGB	9	9	9					5

-- Section 3.8.x, Shared Exponent Texture Color Conversion

Insert this section AFTER section 3.8.14 Texture Comparison Modes and BEFORE section 3.8.15 Texture Application (and after the "sRGB Texture Color Conversion" if EXT_texture_sRGB is supported).

"If the currently bound texture's internal format is RGB9_E5_EXT, the red, green, blue, and shared bits are converted to color components (prior to filtering) using the following shared exponent decoding.

The components red_s, green_s, blue_s, and exp_shared values (see section 3.8.1) are treated as unsigned integers and are converted to red, green, blue as follows:

```
   red   = red_s   * 2^(exp_shared - B)
   green = green_s * 2^(exp_shared - B)
   blue  = blue_s  * 2^(exp_shared - B)"
```

Additions to Chapter 4 of the 2.0 Specification (Per-Fragment Operations and the Frame Buffer)

-- Section 4.3.2, Reading Pixels

Add a row to table 4.7 (page 224);

type Parameter	GL Data Type	Component Conversion Formula
UNSIGNED_INT_5_9_9_9_REV_EXT	uint	special

    Replace second paragraph of "Final Conversion" (page 222) to read:

    For an RGBA color, if <type> is not FLOAT or
    UNSIGNED_INT_5_9_9_9_REV_EXT, or if the CLAMP_READ_COLOR_ARB is
    TRUE, or CLAMP_READ_COLOR_ARB is FIXED_ONLY_ARB and the selected
    color (or texture) buffer is a fixed-point buffer, each component
    is first clamped to [0,1]. Then the appropriate conversion formula
    from table 4.7 is applied the component.

    In the special case when calling ReadPixels with a type of
    UNSIGNED_INT_5_9_9_9_REV_EXT and format of RGB, the conversion
    is done as follows: The returned data are packed into a series of
    GL uint values. The red, green, and blue components are converted
    to red_s, green_s, blue_s, and exp_shared integers as described in
    section 3.8.1 when the internalformat is RGB9_E5_EXT. The red_s,
    green_s, blue_s, and exp_shared are then packed as the 1st, 2nd,
    3rd, and 4th components of the UNSIGNED_INT_5_9_9_9_REV_EXT format
    as shown in table 3.11."

Additions to Chapter 5 of the 2.0 Specification (Special Functions)

    None

Additions to Chapter 6 of the 2.0 Specification (State and State Requests)

-- Section 6.1.3, Enumerated Queries

    Add TEXTURE_SHARED_SIZE_EXT to the list of queries in the first
    sentence of the fifth paragraph (page 247) so it reads:

    "For texture images with uncompressed internal formats, queries of
    value of TEXTURE_RED_SIZE, TEXTURE_GREEN_SIZE, TEXTURE_BLUE_SIZE,
    TEXTURE_ALPHA_SIZE, TEXTURE_LUMINANCE_SIZE, TEXTURE_DEPTH_SIZE,
    TEXTURE_SHARED_SIZE_EXTT, and TEXTURE_INTENSITY_SIZE return the
    actual resolutions of the stored image array components, not the
    resolutions specified when the image array was defined."

Additions to the OpenGL Shading Language specification

    None

Additions to the GLX Specification

    None

GLX Protocol

    None.

Dependencies on ARB_color_buffer_float

    If ARB_color_buffer_float is not supported, replace this amended
    sentence from 4.3.2 above

    "For an RGBA color, if <type> is not FLOAT or
    UNSIGNED_INT_5_9_9_9_REV_EXT, or if the CLAMP_READ_COLOR_ARB is TRUE, or
    CLAMP_READ_COLOR_ARB is FIXED_ONLY_ARB and the selected color buffer

```
    (or texture image for GetTexImage) is a fixed-point buffer (or texture
    image for GetTexImage), each component is first clamped to [0,1]."

    with

    "For an RGBA color, if <type> is not FLOAT or
    UNSIGNED_INT_5_9_9_9_REV_EXT and the selected color buffer (or
    texture image for GetTexImage) is a fixed-point buffer (or texture
    image for GetTexImage), each component is first clamped to [0,1]."
```

Dependencies on EXT_framebuffer_object

```
    If EXT_framebuffer_object is not supported, then
    RenderbufferStorageEXT is not supported and the RGB9_E5_EXT
    internalformat is therefore not supported by RenderbufferStorageEXT.
```

Errors

```
    Relaxation of INVALID_ENUM errors
    ---------------------------------

    TexImage1D, TexImage2D, TexImage3D, CopyTexImage1D, CopyTexImage2D,
    and RenderbufferStorageEXT accept the new RGB9_E5_EXT token for
    internalformat.

    DrawPixels, ReadPixels, TexImage1D, TexImage2D, GetTexImage,
    TexImage3D, TexSubImage1D, TexSubImage2D, TexSubImage3D,
    GetHistogram, GetMinmax, ConvolutionFilter1D, ConvolutionFilter2D,
    ConvolutionFilter3D, GetConvolutionFilter, SeparableFilter2D,
    GetSeparableFilter, ColorTable, ColorSubTable, and GetColorTable
    accept the new UNSIGNED_INT_5_9_9_9_REV_EXT token for type.

    GetTexLevelParameterfv and GetTexLevelParameteriv accept the new
    TEXTURE_SHARED_SIZE_EXT token for <pname>.

    New errors
    ----------

    INVALID_OPERATION is generated by DrawPixels, ReadPixels, TexImage1D,
    TexImage2D, GetTexImage, TexImage3D, TexSubImage1D, TexSubImage2D,
    TexSubImage3D, GetHistogram, GetMinmax, ConvolutionFilter1D,
    ConvolutionFilter2D, ConvolutionFilter3D, GetConvolutionFilter,
    SeparableFilter2D, GetSeparableFilter, ColorTable, ColorSubTable,
    and GetColorTable if <type> is UNSIGNED_INT_5_9_9_9_REV_EXT
    and <format> is not RGB.
```

New State

```
    In table 6.17, Textures (page 278), increment the 42 in "n x Z42*"
    by 1 for the RGB9_E5_EXT format.

    [NOTE: The OpenGL 2.0 specification actually should read "n x Z48*"
    because of the 6 generic compressed internal formats in table 3.18.]
```

    Add the following entry to table 6.17:

Get Value	Type	Get Command	Value	Description	Sec.	Attribute
TEXTURE_SHARED_SIZE_EXT	n x Z+	GetTexLevelParameter	0	xD texture image i's shared exponent field size	3.8	-

New Implementation Dependent State

    None

Appendix

    This source code provides ANSI C routines. It assumes the C "float"
    data type is stored with the IEEE 754 32-bit floating-point format.
    Make sure you define __LITTLE_ENDIAN or __BIG_ENDIAN appropriate
    for your target system.

    XXX: code below not tested on big-endian platform...

------------------- start of source code -----------------------

```
#include <assert.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#define __LITTLE_ENDIAN  1
#define __BIG_ENDIAN     2

#ifdef _WIN32
#define __BYTE_ORDER __LITTLE_ENDIAN
#endif

#define RGB9E5_EXPONENT_BITS          5
#define RGB9E5_MANTISSA_BITS          9
#define RGB9E5_EXP_BIAS               15
#define RGB9E5_MAX_VALID_BIASED_EXP   31

#define MAX_RGB9E5_EXP               (RGB9E5_MAX_VALID_BIASED_EXP - RGB9E5_EXP_BIAS)
#define RGB9E5_MANTISSA_VALUES       (1<<RGB9E5_MANTISSA_BITS)
#define MAX_RGB9E5_MANTISSA          (RGB9E5_MANTISSA_VALUES-1)
#define MAX_RGB9E5        ((float)MAX_RGB9E5_MANTISSA)/RGB9E5_MANTISSA_VALUES * (1<<MAX_RGB9E5_EXP))
#define EPSILON_RGB9E5          ((1.0/RGB9E5_MANTISSA_VALUES) / (1<<RGB9E5_EXP_BIAS))

typedef struct {
#ifdef __BYTE_ORDER
#if __BYTE_ORDER == __BIG_ENDIAN
  unsigned int negative:1;
  unsigned int biasedexponent:8;
  unsigned int mantissa:23;
#elif __BYTE_ORDER == __LITTLE_ENDIAN
  unsigned int mantissa:23;
  unsigned int biasedexponent:8;
  unsigned int negative:1;
#endif
#endif
} BitsOfIEEE754;

typedef union {
  unsigned int raw;
  float value;
  BitsOfIEEE754 field;
} float754;
```

```
typedef struct {
#ifdef __BYTE_ORDER
#if __BYTE_ORDER == __BIG_ENDIAN
  unsigned int biasedexponent:RGB9E5_EXPONENT_BITS;
  unsigned int b:RGB9E5_MANTISSA_BITS;
  unsigned int g:RGB9E5_MANTISSA_BITS;
  unsigned int r:RGB9E5_MANTISSA_BITS;
#elif __BYTE_ORDER == __LITTLE_ENDIAN
  unsigned int r:RGB9E5_MANTISSA_BITS;
  unsigned int g:RGB9E5_MANTISSA_BITS;
  unsigned int b:RGB9E5_MANTISSA_BITS;
  unsigned int biasedexponent:RGB9E5_EXPONENT_BITS;
#endif
#endif
} BitsOfRGB9E5;

typedef union {
  unsigned int raw;
  BitsOfRGB9E5 field;
} rgb9e5;

float ClampRange_for_rgb9e5(float x)
{
  if (x > 0.0) {
    if (x >= MAX_RGB9E5) {
      return MAX_RGB9E5;
    } else {
      return x;
    }
  } else {
    /* NaN gets here too since comparisons with NaN always fail! */
    return 0.0;
  }
}

float MaxOf3(float x, float y, float z)
{
  if (x > y) {
    if (x > z) {
      return x;
    } else {
      return z;
    }
  } else {
    if (y > z) {
      return y;
    } else {
      return z;
    }
  }
}

/* Ok, FloorLog2 is not correct for the denorm and zero values, but we
   are going to do a max of this value with the minimum rgb9e5 exponent
   that will hide these problem cases. */
int FloorLog2(float x)
{
  float754 f;

  f.value = x;
  return (f.field.biasedexponent - 127);
}

int Max(int x, int y)
{
  if (x > y) {
    return x;
  } else {
    return y;
  }
}
```

```
rgb9e5 float3_to_rgb9e5(const float rgb[3])
{
  rgb9e5 retval;
  float maxrgb;
  int rm, gm, bm;
  float rc, gc, bc;
  int exp_shared;
  double denom;

  rc = ClampRange_for_rgb9e5(rgb[0]);
  gc = ClampRange_for_rgb9e5(rgb[1]);
  bc = ClampRange_for_rgb9e5(rgb[2]);

  maxrgb = MaxOf3(rc, gc, bc);
  exp_shared = Max(-RGB9E5_EXP_BIAS-1, FloorLog2(maxrgb)) + 1 + RGB9E5_EXP_BIAS;
  assert(exp_shared <= RGB9E5_MAX_VALID_BIASED_EXP);
  assert(exp_shared >= 0);
  /* This pow function could be replaced by a table. */
  denom = pow(2, exp_shared - RGB9E5_EXP_BIAS - RGB9E5_MANTISSA_BITS);

  rm = (int) floor(rc / denom + 0.5);
  gm = (int) floor(gc / denom + 0.5);
  bm = (int) floor(bc / denom + 0.5);

  assert(rm <= MAX_RGB9E5_MANTISSA);
  assert(gm <= MAX_RGB9E5_MANTISSA);
  assert(bm <= MAX_RGB9E5_MANTISSA);
  assert(rm >= 0);
  assert(gm >= 0);
  assert(bm >= 0);

  retval.field.r = rm;
  retval.field.g = gm;
  retval.field.b = bm;
  retval.field.biasedexponent = exp_shared;

  return retval;
}

void rgb9e5_to_float3(rgb9e5 v, float retval[3])
{
  int exponent = v.field.biasedexponent - RGB9E5_EXP_BIAS - RGB9E5_MANTISSA_BITS;
  float scale = (float) pow(2, exponent);

  retval[0] = v.field.r * scale;
  retval[1] = v.field.g * scale;
  retval[2] = v.field.b * scale;
}

------------------- end of source code -----------------------
```

Issues

1) *What should this extension be called?*

RESOLVED: EXT_texture_shared_exponent

The "EXT_texture" part indicates the extension is in the texture domain and "shared_exponent" indicates the extension is adding a new shared exponent formats.

EXT_texture_rgb9e5 was considered but there's no precedent for extension names to be so explicit (or cryptic?) about format specifics in the extension name.

2) There are many possible encodings for a shared exponent format. Which encoding does this extension specify?

RESOLVED: A single 5-bit exponent stored as an unsigned value biased by 15 and three 9-bit mantissas for each of 3 components. There are no sign bits so all three components must be non-negative. The fractional mantissas assume an implied 0 left of the decimal point because having an implied leading 1 is inconsistent with sharing the exponent. Neither Infinity nor Not-a-Number (NaN) are representable in this shared exponent format.

We chose this format because it closely matches the range and precision of the s10e5 half-precision floating-point described in the ARB_half_float_pixel and ARB_texture_float specifications.

3) Why not an 8-bit shared exponent?

RESOLVED: Greg Ward's RGBE shared exponent encoding uses an 8-bit exponent (same as a single-precision IEEE value) but we believe the rgb9e5 is more generally useful than rgb8e8.

An 8-bit exponent provides far more range than is typically required for graphics applications. However, an extra bit of precision for each component helps in situations where a high magnitude component dominates a low magnitude component. Having an 8-bit shared exponent and 8-bit mantissas are amenable to CPUs that facilitate 8-bit sized reads and writes over non-byte aligned fields, but GPUs do not suffer from this issue.

Indeed GPUs with s10e5 texture filtering can use that same filtering hardware for rgb9e5 textures.

However, future extensions could add other shared exponent formats so we name the tokens to indicate the

4) Should there be an external format and type for rgb9e5?

RESOLVED: Yes, hence the external format GL_RGB9_E5_EXT and type GL_UNSIGNED_INT_5_9_9_9_REV_EXT. This makes it fast to load GL_RGB9_E5_EXT textures without any translation by the driver.

5) Why is the exponent bias 15?

RESOLVED: The best technical choice of 15. Hopefully, this discussion sheds insight into the numerics of the shared exponent format in general.

With conventional floating-point formats, the number corresponding to a finite, non-denorm, non-zero floating-point value is

```
    value = -1^sgn * 2^(exp-bias) * 1.frac
```

where sgn is the sign bit (so 1 for sgn negative because -1^-1 == -1 and 0 means positive because -1^0 == +1), exp is an (unsigned) BIASED exponent and bias is the format's constant bias to subtract to get the unbiased (possibly negative) exponent;

and frac is the fractional portion of the mantissa with the "1." indicating an implied leading 1.

An exp value of zero indicates so-called denormalized values (denorms). With conventional floating-point formats, the number corresponding to a denorm floating-point value is

```
    value = -1^sgn * 2^(exp-bias+1) * 0.frac
```

where the only difference between the denorm and non-denorm case is the bias is one greater in the denorm case and the implied leading digit is a zero instead of a one.

Ideally, the rgb9e5 shared exponent format would represent roughly the same range of finite values as the s10e5 format specified by the ARB_texture_float extension. The s10e5 format has an exponent bias of 15.

While conventional floating-point formats cleverly use an implied leading 1 for non-denorm, finite values, a shared exponent format cannot use an implied leading 1 because each component may have a different magnitude for its most-significant binary digit. The implied leading 1 assumes we have the flexibility to adjust the mantissa and exponent together to ensure an implied leading 1. That flexibility is not present when the exponent is shared.

So the rgb9e5 format cannot assume an implied leading one. Instead, an implied leading zero is assumed (much like the conventional denorm case).

The rgb9e5 format eliminate support representing negative, Infinite, not-a-number (NaN), and denorm values.

We've already discussed how the BIASED zero exponent is used to encode denorm values (and zero) with conventional floating-point formats. The largest BIASED exponent (31 for s10e5, 127 for s23e8) indicates Infinity and NaN values. This means these two extrema exponent values are "off limits" for run-of-the-mill values.

The numbers corresponding to a shared exponent format value are:

```
    value_r = 2^(exp-bias) * 0.frac_r
    value_g = 2^(exp-bias) * 0.frac_g
    value_b = 2^(exp-bias) * 0.frac_b
```

where there is no sgn since all values are non-negative, exp is the (unsigned) BIASED exponent and bias is the format's constant bias to subtract to get the unbiased (possibly negative) exponent; and frac_r, frac_g, and frac_b are the fractional portion of the mantissas of the r, g, and b components respectively with "0." indicating an implied leading 0.

There should be no "off limits" exponents for the shared exponent format since there is no requirement for representing Infinity or NaN values and denorm is not a special case. Because of the implied leading zero, any component with all zeros for its mantissa is zero, no matter the shared exponent's value.

So the run-of-the-mill BIASED range of exponents for s10e5 is 1 to 30. But the rgb9e5 shared exponent format consistently uses the same rule for all exponents from 0 to 31.

What exponent bias best allows us to represent the range of s10e5 with the rgb9e5 format? 15.

Consider the maximum representable finite s10e5 magnitude. The exponent would be 30 (31 would encode an Infinite or NaN value) and the binary mantissa would be 1 followed by ten fractional 1's. Effectively:

```
    s10e5_max  =  1.1111111111 * 2^(30-15)
               =  1.1111111111 * 2^15
```

For an rgb9e5 value with a bias of 15, the largest representable value is:

```
    rgb9e5_max =  0.111111111  * 2^(31-15)
               =  0.111111111  * 2^16
               =  1.11111111   * 2^15
```

If you ignore two LSBs, these values are nearly identical. The rgb9e5_max value is exactly representable as an s10e5 value.

For an rgb9e5 value with a bias of 15, the smallest non-zero representable value is:

```
    rgb9e5_min =  0.000000001  * 2^(0-15)
    rgb9e5_min =  0.000000001  * 2^-15
    rgb9e5_min =  0.0000000001 * 2^-14
```

So the s10e5_min and rgb9e5_min values exactly match (of course, this assumes the shared exponent bias is 15 which might not be the case if other components demand higher exponents).

8) *Should there be an rgb9e5 framebuffer format?*

RESOLVED: No. Rendering to rgb9e5 is better left to another extension and would require the hardware to convert from a (floating-point) RGBA value into an rgb9e5 encoding.

Interactions with EXT_framebuffer_object are specified, but the expectation is this is not a renderable format and glCheckFramebufferStatusEXT would return GL_FRAMEBUFFER_UNSUPPORTED_EXT.

An implementation certainly could make this texture internal format renderable when used with a framebuffer object. Note that the shared exponent means masked components may be lossy in their masking. For example, a very small but non-zero value in a masked component could get flushed to zero if a large enough value is written into an unmasked component.

9) *Should automatic mipmap generation be supported for rgb9e5 textures?*

RESOLVED: Yes.

10) *Should non-texture and non-framebuffer commands for loading pixel data accept the GL_UNSIGNED_INT_5_9_9_9_REV_EXT type?*

RESOLVED: Yes.

Once the pixel path has to support the new type/format combination of GL_UNSIGNED_INT_5_9_9_9_REV_EXT / GL_RGB for specifying and querying texture images, it might as well be supported for all commands that pack and unpack RGB pixel data.

The specification is written such that the glDrawPixels type/format parameters are accepted by glReadPixels, glTexGetImage, glTexImage2D, and other commands that are specified in terms of glDrawPixels.

11) *Should non-texture internal formats (such as for color tables, convolution kernels, histogram bins, and min/max tables) accept GL_RGB9_E5_EXT format?*

RESOLVED: No.

That's pointless. No hardware is ever likely to support GL_RGB9_E5_EXT internalformats for anything other than textures and maybe color buffers in the future. This format is not interesting for color tables, convolution kernels, etc.

12) *Should a format be supported with sign bits for each component?*

RESOLVED: No.

An srgb8e5 format with a sign bit per component could be useful but is better left to another extension.

13) *The rgb9e5 allows two 32-bit values encoded as rgb9e5 to correspond to the exact same 3 components when expanded to floating-point. Is this a problem?*

RESOLVED: No, there's no problem here.

An encoder is likely to always pack components so at least one mantissa will have an explicit leading one, but there's no requirement for that.

Applications might be able to take advantage of this by quickly dividing all three components by a power-of-two by simply subtracting log2 of the power-of-two from the shared exponent (as long as the exponent is greater than zero prior to the subtract).

Arguably, the shared exponent format could maintain a slight amount of extra precision (one bit per mantissa) if the format said if the most significant bits of all three mantissas are either all one or all zero and the biased shared exponent was not zero, then an implied leading 1 should be assumed and the shared exponent should be treated as one smaller than it really is. While this would preserve an extra least-significant bit of mantissa precision for components of approximately the same magnitude, it would complicate the encoding and decoding of shared exponent values.

14) *Can you provide some C code for encoding three floating-point values into the rgb9e5 format?*

RESOLVED: Sure. See the Appendix.

15) *Should we support a non-REV version of the GL_UNSIGNED_INT_5_9_9_9_REV_EXT token?*

RESOLVED: No. The shared exponent is always the 5 most significant bits of the 32 bit word. The first (red) mantissa is in the least significant 9 bits, followed by 9 bits for the second (green) mantissa, followed by 9 bits for the third (blue) mantissa. We don't want to promote different arrangements of the bitfields for rgb9e5 values.

16) *Can you use the GL_UNSIGNED_INT_5_9_9_9_REV_EXT format with just any format?*

RESOLVED: You can only use the GL_UNSIGNED_INT_5_9_9_9_REV_EXT format with GL_RGB. Otherwise, the GL generates an GL_INVALID_OPERATION error. Conceptually, GL_UNSIGNED_INT_5_9_9_9_REV_EXT is a 3-component format that just happens to have 5 shared bits too. Just as the GL_UNSIGNED_BYTE_3_3_2 format just works with GL_RGB (or else the GL generates an GL_INVALID_OPERATION error), so should GL_UNSIGNED_INT_5_9_9_9_REV_EXT.

17) *What should GL_TEXTURE_SHARED_SIZE_EXT return when queried with GetTexLevelParameter?*

RESOLVED: Return 5 for the RGB9_E5_EXT internal format and 0 for all other existing formats.

This is a count of the number of bits in the shared exponent.

18) *What should GL_TEXTURE_RED_SIZE, GL_TEXTURE_GREEN_SIZE, and GL_TEXTURE_BLUE_SIZE return when queried with GetTexLevelParameter for a GL_RGB9_E5_EXT texture?*

RESOLVED: Return 9 for each.

Revision History

None

Name

    EXT_texture_sRGB

Name Strings

    GL_EXT_texture_sRGB

Contributors

    Alain Bouchard, Matrox
    Brian Paul, Tungsten Graphics
    Daniel Vogel, Epic Games
    Eric Werness, NVIDIA
    Kiril Vidimce, Pixar
    Mark J. Kilgard, NVIDIA
    Pat Brown, NVIDIA
    Yanjun Zhang, S3 Graphics
    Jeremy Sandmel, Apple
    Herb Kuta, Quantum3D

Contact

    Mark J. Kilgard, NVIDIA Corporation (mjk 'at' nvidia.com)

Status

    Implemented by NVIDIA's Release 80 driver series for GeForce FX
    (NV3x), GeForce 6 and 7 Series (NV4x and G7x), and Quadro FX (NV3xGL,
    NV4xGL, G7xGL).

Version

    Date: January 24, 2007
    Revision: 0.8

Number

    315

Dependencies

    OpenGL 1.1 required

    EXT_texture_compression_s3tc interacts with this extension.

    NV_texture_compression_vtc interacts with this extension.

    This extension is written against the OpenGL 2.0 (September 7,
    2004) specification.

Overview

    Conventional texture formats assume a linear color space. So for
    a conventional internal texture format such as GL_RGB8, the 256
    discrete values for each 8-bit color component map linearly and
    uniformly to the [0,1] range.

The sRGB color space is based on typical (non-linear) monitor characteristics expected in a dimly lit office. It has been standardized by the International Electrotechnical Commission (IEC) as IEC 61966-2-1. The sRGB color space roughly corresponds to 2.2 gamma correction.

This extension adds a few new uncompressed and compressed color texture formats with sRGB color components.

Issues

1) *What should this extension be called?*

   RESOLVED: EXT_texture_sRGB.

   The "EXT_texture" part indicates the extension is in the texture domain and "sRGB" indicates the extension is adding a set of sRGB formats. ARB_texture_float is similarly named where "_float" indicates float texture formats are added by the extension.

   The mixed-case spelling of sRGB is the established usage so "_sRGB" is preferred to "_srgb". The "s" stands for standard (color space).

   For token names, we use "SRGB" since token names are uniformly capitalized.

2) *Should this extension mandate that sRGB conversion be performed pre-filtering?*

   RESOLVED: Post-filtering sRGB color conversion is allowed though pre-filtering conversion is the preferred approach.

   Ideally, sRGB conversion moves from the non-linear sRGB to the linear RGB color space. However, implementations should be provided leeway as to whether sRGB conversion occurs before or after texture filtering of RGB components.

3) *Should the alpha component of sRGB texture formats be gamma-corrected?*

   RESOLVED: No. Alpha is correctly understood to be a weighting factor that is best stored in a linear representation. The alpha component should always be stored as a linear value.

   "SRGB_ALPHA" is used to indicate sRGB formats with an alpha component. This naming (as opposed to something like "SRGBA") helps highlight the fact that the alpha component is separate and stored with a linear distribution of precision.

4) *Should formats for sRGB luminance values be supported?*

   RESOLVED: Yes. Implementations can always support luminance and luminance-alpha sRGB formats as an RGB8 or RGBA8 format with replicated R, G, and B values.

For lack of a better term, "SLUMINANCE" will be used within token names to indicate sRGB values with identical red, green, and blue components.

5) *Should formats for sRGB intensity values be supported?*

RESOLVED: No. Intensity uses the same value for both luminance and alpha. Treating a single value as an sRGB luminance value and a linear alpha value is undesirable.

Hardware design is simplified if alpha never involves sRGB conversions.

6) *Should all component sizes be supported for sRGB components or just 8-bit?*

RESOLVED: Just 8-bit. For sRGB values with more than 8 bit of precision, a linear representation may be easier to work with and adequately represent dim values. Storing 5-bit and 6-bit values in sRGB form is unnecessary because applications sophisticated enough to sRGB to maintain color precision will demand at least 8-bit precision for sRGB values.

Because hardware tables are required sRGB conversions, it doesn't make sense to burden hardware with conversions that are unlikely when 8-bit is the norm for sRGB values.

7) *Should color tables, convolution kernels, histogram table, and minmax table entries support sRGB formats?*

RESOLVED: No.

The internalformat for histogram table entries determines the bit precision of the histogram bin counters so indicating the sRGB color space is meaningless in this context. The internalformat for minmax table entries simply indicates the components for minmax bounding so indicating the sRGB color space is meaningless.

Convolution filter values are weighting factors rather than color values needing a color space.

Color table entries may be colors but the component values are typically stored with more than 8 bits already. For example, software implementations of the OpenGL color table functionality typically store colors in floating-point.

8) *Should generic compressed sRGB formats be supported?*

RESOLVED: Yes. Implementations are free simply to use uncompressed sRGB formats to implement the GL_COMPRESSED_SRGB_* formats.

9) *Should S3TC compressed sRGB formats be supported?*

RESOLVED: Yes, but only if EXT_texture_compression_s3tc is also advertised. For competitive reasons, we expect OpenGL will need an S3TC-based block compression format for sRGB data.

Rather than expose a separate "sRGB_compression" extension, it makes more sense to specify a dependency between EXT_texture_compression_s3tc and this extension such that when BOTH extensions are exposed, the GL_COMPRESSED_SRGB*_S3TC_DXT*_EXT tokens are accepted.

We avoid explicitly requiring S3TC formats when EXT_texture_sRGB is advertised to avoid IP encumbrances.

10) *Should the S3TC decompression algorithm be affected by support for sRGB component values?*

RESOLVED: No.

S3TC involves the linear weighting of two per-block R5G6B5 colors. The sRGB to linear RGB color conversion should occur AFTER the linear weighting of the two per-block colors performed during texel decompression.

Also be aware that an sRGB value with 8-bit red, green, and blue components must be quantized to a 5, 6, and 5 bits respectively to form the two per-block R5G6B5 colors.

S3TC compressors may wish to account for the sRGB color space as part of the compression algorithm.

11) *Should VTC compressed sRGB formats be supported?*

RESOLVED. Yes, for the same reasons as S3TC.

12) *Should pixel data entering or exiting the OpenGL pixel path be labeled as sRGB or conventional linear RGB? This would allow pixels labeled as sRGB to be converted to a linear RGB color space prior to processing by the pixel path which includes operations such as convolution, scale, and bias that presume a linear color space. If the destination (say a texture with an sRGB internal format) was sRGB, then linear RGB components would be converted to sRGB prior to being packed into the texture image. This would assume new format parameters to glDrawPixels and glReadPixels indicating the source or destination format was sRGB if a GL_SRGB_EXT or GL_SRGB_ALPHA_EXT format is specified. Likewise, a format parameter to glTexImage2D such as GL_SRGB_EXT would indicate the pixel data was already in an sRGB color space where GL_RGB would indicate a linear color space. New state would indicate if the framebuffer held sRGB or linear RGB pixels.*

RESOLVED: No.

The pixel path should be left blind to color spaces and provide no implicit conversions.

Core pixel maps and ARB_imaging provides sufficient color tables so that applications interested in managing color space conversions within the pixel path can do so themselves.

A 256 entry table outputting floating-point values is sufficient to convert sRGB to linear RGB.

However when converting from linear RGB to sRGB, one must be careful to make sure the source linear RGB values are specified with more than 8 bits of precision and the color table to implement the conversion must likewise have more than 256 entries. A power-of-two table sufficient to map values to each of the 256 sRGB encodings for an 8-bit sRGB component requires at least 4096 entries (a fairly large color table).

Because vertex and fragment programs and shaders operate in floating-point and have sufficient programmability to implement the sRGB to linear RGB and vice versa without resorting to large tables.

13) *Does this extension imply filtered results from sRGB texture have more than 8 bits of precision?*

RESOLVED: Effectively, yes.

8-bit components of sRGB texels are converted to linear RGB values which requires more than 8 bits to avoid lose of precision. This implies the filtering involve more than 8 bits of color precision per component. Moreover, fragment color (whether by a fragment program, vertex program, or glTexEnv modes) should operate at precision beyond 8 bits per color component.

The exact precision maintained (and its distribution) is left to implementations to define but returning at least 12 but more likely 16 linear bits per component, post-filtering, is a reasonable expectation for developers.

This extension assumes fragment coloring is performed

14) *What must be specified as far as how do you convert to and from sRGB and linear RGB color spaces?*

RESOLVED: The specification language needs to only supply the sRGB to linear RGB conversion (see section 3.8.x below).

For completeness, the accepted linear RGB to sRGB conversion (the inverse of the function specified in section 3.8.x) is as follows:

Given a linear RGB component, cl, convert it to an sRGB component, cs, in the range [0,1], with this pseudo-code:

```
if (isnan(cl)) {
    /* Map IEEE-754 Not-a-number to zero. */
    cs = 0.0;
} else if (cl > 1.0) {
    cs = 1.0;
} else if (cl < 0.0) {
    cs = 0.0;
} else if (cl < 0.0031308) {
    cs = 12.92 * cl;
} else {
    cs = 1.055 * pow(cl, 0.41666) - 0.055;
}
```

sRGB components are typically stored as unsigned 8-bit fixed-point values. If cs is computed with the above pseudo-code, cs can be converted to a [0,255] integer with this formula:

```
csi = floor(255.0 * cs + 0.5)
```

15) *Does this extension provide any sort of sRGB framebuffer formats or guarantee images rendered with sRGB textures will "look good" when output to a device supporting an sRGB color space?*

RESOLVED: No.

Whether the displayed framebuffer is displayed to a monitor that faithfully reproduces the sRGB color space is beyond the scope of this extension. This involves the gamma correction and color calibration of the physical display device.

With this extension, artists can author content in an sRGB color space and provide that sRGB content for use as texture imagery that can be properly converted to linear RGB and filtered as part of texturing in a way that preserves the sRGB distribution of precision, but that does NOT mean sRGB pixels are output to the framebuffer. Indeed, this extension provides texture formats that convert sRGB to linear RGB as part of filtering.

With programmable shading, an application could perform a linear RGB to sRGB conversion just prior to emitting color values from the shader. Even so, OpenGL blending (other than simple modulation) will perform linear math operations on values stored in a non-linear space which is technically incorrect for sRGB-encoded colors.

One way to think about these sRGB texture formats is that they simply provide color components with a distribution of values distributed to favor precision towards 0 rather than evenly distributing the precision with conventional non-sRGB formats such as GL_RGB8.

16) How does this extension interact with EXT_framebuffer_object?

RESOLVED: No specific interaction language is necessary but there is no provision that pixels written into a framebuffer object with a texture with an sRGB internal format for its color buffer will in anyway convert the output color values into an sRGB color space. A fragment program or shader could be written to convert linear RGB values to sRGB values prior to shader output, but NO automatic conversion is performed.

So you can create a texture with an sRGB internal format (such as GL_SRGB8_ALPHA8_EXT), bind that texture to a framebuffer object with glFramebufferTexture2DEXT, and then render into that framebuffer. If you then texture with the sRGB texture, the texels within the texture are treated as sRGB values for filtering.

17) Should sRGB be supported with a texture parameter rather than new texture formats?

RESOLVED: Adding new texture formats is the right approach.

Hardware is expected to implements sRGB conversions via hardwired look-up tables. Such tables are expensive (when sRGB isn't being used, they are basically "wasted gates") and so we want to minimize the number of unique tables that hardware must support. However OpenGL supports various component sizes for RGB and RGBA textures.

Various RGB texture formats have different bit sizes for R, G, and B that map to [0,1]. Think about RGB5. It encodes values 0/15, 1/15, 2/15, ... 14/15, and 15/15. Excepting 0/15==0.0 and 15/15==1.0, those values are different than the values for RGB8 which would be 0/255, 1/255, ... 254/255, 255/255. Technically, you'd need a different sRGB table to toggle between RGB4 and sRGB4 than you'd need to toggle between RGB8 and sRGB8. There are also RGB12 and RGB16 textures where it is simply not tractable to implement 4096 and 65,536 entry tables, nor is the "real" sRGB conversion math cheap enough to evaluate directly at those precisions.

What this extension shouldn't require is sRGB conversion for any component sizes beyond 8-bit. Indeed, it appears the only component sizes sRGB users really care about are 8-bit components. This is because if you have more than 8 bits per component, you typically have enough precision to avoid the complexity created by a non-linear RGB component encoding. Additionally, sRGB users are picky about color reproduction so fewer than 8 bits is generally not acceptable to them.

The problem with making a "toggle" (say controlled by glTexParameter) is that hardware would very likely (indeed it's pretty much certain) not implement toggling between RGB12 and sRGB12 formats. Recall that OpenGL doesn't mandate internal formats so you can request GL_RGB8 and have the implementation actually given you RGB12 or RGB10 or R5G6B5.

It is inappropriate to put in a texture parameter mode where we say "this mode works just with GL_RGB8 and GL_RGBA8 and yet only when the underlying internal format is actually RGB8 or RGBA8". We'd also surely preclude floating-point RGB formats, signed RGB formats, new HDR formats, and certain compressed RGB formats from being included because such formats don't really even make sense for sRGB.

By adding new formats specifically for the sRGB color space, we avoid all these problems.

We also avoid an awkward precedent where other more varied color spaces (CYMK, XYZ, and YUV being obvious examples) have to "toggle" between RGB and RGBA formats. Indeed, already extensions for such other color spaces (YUV and CMYK at least) set the precedent of introducing new texture formats.

18) How is the texture border color handled for sRGB formats?

RESOLVED: The texture border color is specified as four floating-point values. Given that the texture border color can be specified at such high precision, it is always treated as a linear RGBA value.

Only texel components are converted from the sRGB encoding to a linear RGB value ahead of texture filtering. The border color can be used "as is" without any conversion.

The implication of this is, for example, that two textures with GL_RGBA8 and GL_SRGB8_ALPHA8_EXT internal formats respectively and a border color of (0.4, 0.2, 0.9, 0.1) and the GL_CLAMP_TO_BORDER wrap mode will both return (0.4, 0.2, 0.9, 0.1) if 100% of the border color is sampled.

By keeping the texture border color specified as a linear RGB value at the API level allows developers to specify the high-precision texture border color in a single consistent color space without concern for how the sRGB conversion is implemented in relation to filtering.

An implementation that does post-filtering sRGB conversion is likely to store convert the texture border color to sRGB within the driver so it can be filtered with the sRGB values coming from texels and then the filtered sRGB value is converted to linear RGB.

By maintaining the texture border color always in linear RGB, we avoid developers having to know if an implementation is performing the sRGB conversion (ideally) pre-filtering or (less ideally) post-filtering.

19) How does this extension interact with NV_texture_expand_normal?

RESOLVED: sRGB components are not affected by the "expand normal" mode even though they are unsigned components because they have non-linear precision (similar to floating-point).

The alpha component of GL_SRGB8_ALPHA8_EXT and other sRGB formats
with an alpha component is affected by the "expand normal" mode.

The sRGB formats have unsigned components with [0,1] range which
is the requirement for the NV_texture_expand_normal extension's
operation.

Be warned because sRGB formats distribute their precision more
towards zero, enabling the GL_EXPAND_NORMAL_NV mode with sRGB
textures will mean there are more representable negative values
than positive values. For example, the 8-bit value 128 maps
roughly to zero when encoded with a GL_RGB8 internal format and
then remapped with the GL_EXPAND_NORMAL_NV mode. In contrast,
the sRGB encoded 8-bit value 188 maps roughly to zero when encoded
with a GL_SRGB8_ALPHA8 internal format and then remapped with
GL_EXPAND_NORMAL_NV. Still 0 will map to -1 and 255 will map
to +1 in either case.

20) *What values should glGetTexImage return? Are the sRGB values
returned "as-is" or are they converted to linear RGB first?*

RESOLVED: sRGB values are returned "as-is" without an
sRGB-to-linear conversion. Unlike other commands that transfer
pixel data, "No pixel transform operations are performed" on
the queried texture image.

21) *How does glCopyTex[Sub]Image work with sRGB? Suppose we're
rendering to a floating point pbuffer or framebuffer object and
do CopyTexImage. Are the linear framebuffer values converted
to sRGB during the copy?*

RESOLVED: No, linear framebuffer values will NOT be automatically
converted to the sRGB encoding during the copy. If such a
conversion is desired, as explained in issue 12, the red, green,
and blue pixel map functionality can be used to implement a
linear-to-sRGB encoding translation.

22) *Should the new COMPRESSED_SRGB_* formats be listed in an
implementation's GL_COMPRESSED_TEXTURE_FORMATS list?*

RESOLVED: No. Section 3.8.1 says formats listed by
GL_COMPRESSED_TEXTURE_FORMATS are "suitable for general-purpose
usage." The non-linear distribution of red, green, and
blue for these sRGB compressed formats makes them not really
general-purpose.

23) *Could this extension be implemented by hardware with no special
hardware support for sRGB but does support native GL_RGB12 or
GL_RGB16 textures? If so, how?*

RESOLVED. Yes.

The conversion from the sRGB encoding to linear encoding described
in section 3.8.x could be performed at texture specification
time (after the image has been transformed by the pixel path)
rather than texture fetch time.

When glTexImage2D, glTexSubImage2D, glCopyTexImage2D, etc. occur, the pixels would be transformed by the pixel path as normal and then when pixels are converted to the internal texture format, the section 3.8.x conversion is applied to the red, green, and blue components (not alpha). The result of this conversion can be quantized and stored into the respective red, green, or blue 12-bit or 16-bit component of the stored texel.

This means when a texture fetch occurs, no fetch-time conversion is required.

The advantages of this approach is that sRGB conversion is pre-filtering (the ideal) and the hardware is not required to have texture fetch hardware to perform the special sRGB conversion.

The disadvantage of this technique is that sRGB textures may require more space than required if 8-bit component sRGB components are stored in texture memory.

The ability to implement this extension in this manner provides one more justification to avoid a "toggle" texture parameter for sRGB conversion or not.

One caveat to this approach is that glGetTexImage should return the texel values with the sRGB conversion from section 3.8.x "reverse converted". (The section 3.8.x function is reversible.) As specified, the conversion is performed at fetch time so the understanding is that data returned by glGetTexImage should be the texels prior to the conversion. If the components are stored converted, that means they must be reverse-converted when returned by glGetTexImage.

24) How should mipmap generation work for sRGB textures?

RESOLVED: The best way to perform mipmap generation for sRGB textures is by downsampling the sRGB image in a linear color space.

This involves converting the RGB components of sRGB texels in a given texture image level to linear RGB space, filtering appropriately in that linear RGB space, and then converting the linear RGB values to sRGB for storage in the downsampled texture level image.

(Remember alpha, when present, is linear even in sRGB texture formats.)

The OpenGL specification says "No particular filter algorithm is required, though a box filter is recommended as the default filter" meaning there is no requirement for how even non-sRGB mipmaps should be generated. So while the resolution to this issue is technically a recommendation, it is however a strongly advised recommendation.

The rationale for why sRGB textures should be converted to linear space prior to filtering and converted back to sRGB after filtering is clear. If an implementation naively simply performed linear filtering on (non-linear) sRGB components as if they were in a linear space, the result tends to be a subtle darkening of the texture images as mipmap generation continues recursively. This darkening is an inappropriate basis that the resolved "best way" above would avoid.

New Procedures and Functions

None

New Tokens

Accepted by the <internalformat> parameter of TexImage1D, TexImage2D, TexImage3D, CopyTexImage1D, CopyTexImage2D:

SRGB_EXT	0x8C40
SRGB8_EXT	0x8C41
SRGB_ALPHA_EXT	0x8C42
SRGB8_ALPHA8_EXT	0x8C43
SLUMINANCE_ALPHA_EXT	0x8C44
SLUMINANCE8_ALPHA8_EXT	0x8C45
SLUMINANCE_EXT	0x8C46
SLUMINANCE8_EXT	0x8C47
COMPRESSED_SRGB_EXT	0x8C48
COMPRESSED_SRGB_ALPHA_EXT	0x8C49
COMPRESSED_SLUMINANCE_EXT	0x8C4A
COMPRESSED_SLUMINANCE_ALPHA_EXT	0x8C4B

Accepted by the <internalformat> parameter of TexImage2D, CopyTexImage2D, and CompressedTexImage2DARB and the <format> parameter of CompressedTexSubImage2DARB:

COMPRESSED_SRGB_S3TC_DXT1_EXT	0x8C4C
COMPRESSED_SRGB_ALPHA_S3TC_DXT1_EXT	0x8C4D
COMPRESSED_SRGB_ALPHA_S3TC_DXT3_EXT	0x8C4E
COMPRESSED_SRGB_ALPHA_S3TC_DXT5_EXT	0x8C4F

Additions to Chapter 2 of the 1.2 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.2 Specification (Rasterization)

-- Section 3.8.1, Texture Image Specification:

Add 4 new rows to Table 3.16 (page 154).

Sized Internal Format	Base Internal Format	R bits	G bits	B bits	A bits	L bits	I bits	D bits
SRGB8_EXT	RGB	8	8	8				
SRGB8_ALPHA8_EXT	RGBA	8	8	8	8			
SLUMINANCE_EXT	LUMINANCE					8		
SLUMINANCE_ALPHA8_EXT	LUMINANCE_ALPHA				8	8		

Add 4 new rows to Table 3.17 (page 155).

Compressed Internal Format	Base Internal Format
COMPRESSED_SRGB_S3TC_DXT1_EXT	RGB
COMPRESSED_SRGB_ALPHA_S3TC_DXT1_EXT	RGBA
COMPRESSED_SRGB_ALPHA_S3TC_DXT3_EXT	RGBA
COMPRESSED_SRGB_ALPHA_S3TC_DXT5_EXT	RGBA

Add 4 new rows to Table 3.18 (page 155).

Generic Compressed Internal Format	Base Internal Format
COMPRESSED_SRGB_EXT	RGB
COMPRESSED_SRGB_ALPHA_EXT	RGBA
COMPRESSED_SLUMINANCE_EXT	LUMINANCE
COMPRESSED_SLUMINANCE_ALPHA_EXT	LUMINANCE_ALPHA

-- Section 3.8.x, sRGB Texture Color Conversion

Insert this section AFTER section 3.8.14 Texture Comparison Modes and BEFORE section 3.8.15 Texture Application.

"If the currently bound texture's internal format is one of SRGB_EXT, SRGB8_EXT, SRGB_ALPHA_EXT, SRGB8_ALPHA8_EXT, SLUMINANCE_ALPHA_EXT, SLUMINANCE8_ALPHA8_EXT, SLUMINANCE_EXT, SLUMINANCE8_EXT, COMPRESSED_SRGB_EXT, COMPRESSED_SRGB_ALPHA_EXT, COMPRESSED_SLUMINANCE_EXT COMPRESSED_SLUMINANCE_ALPHA_EXT, COMPRESSED_SRGB_S3TC_DXT1_EXT, COMPRESSED_SRGB_ALPHA_S3TC_DXT1_EXT, COMPRESSED_SRGB_ALPHA_S3TC_DXT3_EXT, or COMPRESSED_SRGB_ALPHA_S3TC_DXT5_EXT, the red, green, and blue components are converted from an sRGB color space to a linear color space as part of filtering described in sections 3.8.8 and 3.8.9. Any alpha component is left unchanged. Ideally, implementations should perform this color conversion on each sample prior to filtering but implementations are allowed to perform this conversion after filtering (though this post-filtering approach is inferior to converting from sRGB prior to filtering).

The conversion from an sRGB encoded component, cs, to a linear component, cl, is as follows.

```
      {  cs / 12.92,                cs <= 0.04045
   cl = {
      {  ((cs + 0.055)/1.055)^2.4,   cs >  0.04045
```

Assume cs is the sRGB component in the range [0,1]."

Additions to Chapter 4 of the 1.2 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.2 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

    None

Additions to the OpenGL Shading Language specification

    None

Additions to the GLX Specification

    None

Dependencies on ARB_texture_compression and OpenGL 1.3 or later

    If ARB_texture_compression or OpenGL 1.3 or later is NOT supported,
    ignore the new COMPRESSED_* tokens, the additions to tables 3.17
    and 3.18, and the errors associated with the Compressed* commands.

Dependencies on EXT_texture_compression_s3tc

    If EXT_texture_compression_s3tc is NOT supported, ignore the new
    COMPRESSED_*_S3TC_DXT* tokens, the additions to table 3.17, errors
    related to the COMPRESSED_*_S3TC_DXT* tokens, and related discussion.

    Add COMPRESSED_SRGB_S3TC_DXT1_EXT,
    COMPRESSED_SRGB_ALPHA_S3TC_DXT1_EXT,
    COMPRESSED_SRGB_ALPHA_S3TC_DXT3_EXT, and
    COMPRESSED_SRGB_ALPHA_S3TC_DXT5_EXT to token lists in the section
    3.8.2 specification language added by EXT_texture_compression_s3tc
    when the internal formats COMPRESSED_RGB_S3TC_DXT1_EXT,
    COMPRESSED_RGBA_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT3_EXT, and
    COMPRESSED_RGBA_S3TC_DXT5_EXT are listed.

Dependencies on NV_texture_compression_vtc

    If NV_texture_compression_vtc IS supported, allow the following
    tokens to be accepted by the <internalformat> parameter
    of CompressedTexImage3DARB and the <format> parameter of
    CompressedTexSubImage3DARB:

```
        COMPRESSED_SRGB_S3TC_DXT1_EXT
        COMPRESSED_SRGB_ALPHA_S3TC_DXT1_EXT
        COMPRESSED_SRGB_ALPHA_S3TC_DXT3_EXT
        COMPRESSED_SRGB_ALPHA_S3TC_DXT5_EXT
```

GLX Protocol

    None.

Errors

```
    Relaxation of INVALID_ENUM errors
    ---------------------------------
```

    TexImage1D, TexImage2D, TexImage3D, CopyTexImage1D, CopyTexImage2D,
    CompressedTexImage2DARB, CompressedTexSubImage2DARB now accept the
    new tokens as listed in the "New Tokens" section.

    New errors

```
    ----------

    INVALID_OPERATION is generated by CompressedTexImage2DARB if
    if <internalformat> is COMPRESSED_SRGB_S3TC_DXT1_EXT,
    COMPRESSED_SRGB_ALPHA_S3TC_DXT1_EXT,
    COMPRESSED_SRGB_ALPHA_S3TC_DXT3_EXT, or
    COMPRESSED_SRGB_ALPHA_S3TC_DXT5_EXT and <border> is not equal to
    zero.

    INVALID_OPERATION is generated by TexSubImage2D
    CopyTexSubImage2D, or CompressedTexSubImage2D if INTERNAL_FORMAT is
    COMPRESSED_SRGB_S3TC_DXT1_EXT, COMPRESSED_SRGB_ALPHA_S3TC_DXT1_EXT,
    COMPRESSED_SRGB_ALPHA_S3TC_DXT3_EXT, or
    COMPRESSED_SRGB_ALPHA_S3TC_DXT5_EXT and any of the following apply:
    <width> is not a multiple of four or equal to TEXTURE_WIDTH; <height>
    is not a multiple of four or equal to TEXTURE_HEIGHT; <xoffset>
    or <yoffset> is not a multiple of four.

    INVALID_ENUM is generated by CompressedTexImage1DARB if
    <internalformat> is COMPRESSED_SRGB_S3TC_DXT1_EXT,
    COMPRESSED_SRGB_ALPHA_S3TC_DXT1_EXT,
    COMPRESSED_SRGB_ALPHA_S3TC_DXT3_EXT, or
    COMPRESSED_SRGB_ALPHA_S3TC_DXT5_EXT.

    INVALID_ENUM is generated by CompressedTexSubImage1DARB if <format> is
    COMPRESSED_SRGB_S3TC_DXT1_EXT, COMPRESSED_SRGB_ALPHA_S3TC_DXT1_EXT,
    COMPRESSED_SRGB_ALPHA_S3TC_DXT3_EXT, or
    COMPRESSED_SRGB_ALPHA_S3TC_DXT5_EXT.

    Errors if NV_texture_compression_vtc is NOT supported
    -----------------------------------------------------

    INVALID_ENUM is generated by CompressedTexImage3DARB if
    <internalformat> is COMPRESSED_SRGB_S3TC_DXT1_EXT,
    COMPRESSED_SRGB_ALPHA_S3TC_DXT1_EXT,
    COMPRESSED_SRGB_ALPHA_S3TC_DXT3_EXT, or
    COMPRESSED_SRGB_ALPHA_S3TC_DXT5_EXT.

    INVALID_ENUM is generated by CompressedTexSubImage3DARB if <format> is
    COMPRESSED_SRGB_S3TC_DXT1_EXT, COMPRESSED_SRGB_ALPHA_S3TC_DXT1_EXT,
    COMPRESSED_SRGB_ALPHA_S3TC_DXT3_EXT, or
    COMPRESSED_SRGB_ALPHA_S3TC_DXT5_EXT.

    Errors if NV_texture_compression_vtc IS supported
    -----------------------------------------------------

    INVALID_OPERATION is generated by CompressedTexImage3DARB
    if <internalformat> is COMPRESSED_SRGB_S3TC_DXT1_EXT,
    COMPRESSED_SRGB_ALPHA_S3TC_DXT1_EXT,
    COMPRESSED_SRGB_ALPHA_S3TC_DXT3_EXT, or
    COMPRESSED_SRGB_ALPHA_S3TC_DXT5_EXT and <border> is not equal to
    zero.

    INVALID_OPERATION is generated by TexSubImage3D or CopyTexSubImage3D
    if INTERNAL_FORMAT is COMPRESSED_SRGB_S3TC_DXT1_EXT,
    COMPRESSED_SRGB_ALPHA_S3TC_DXT1_EXT,
    COMPRESSED_SRGB_ALPHA_S3TC_DXT3_EXT, or
```

COMPRESSED_SRGB_ALPHA_S3TC_DXT5_EXT and any of the following apply: <width> is not a multiple of four or equal to TEXTURE_WIDTH; <height> is not a multiple of four or equal to TEXTURE_HEIGHT; <xoffset> or <yoffset> is not a multiple of four.

INVALID_OPERATION is generated by CompressedTexSubImage3D if INTERNAL_FORMAT is COMPRESSED_SRGB_S3TC_DXT1_EXT, COMPRESSED_SRGB_ALPHA_S3TC_DXT1_EXT, COMPRESSED_SRGB_ALPHA_S3TC_DXT3_EXT, or COMPRESSED_SRGB_ALPHA_S3TC_DXT5_EXT and any of the following apply: <width> is not a multiple of four or equal to TEXTURE_WIDTH; <height> is not a multiple of four or equal to TEXTURE_HEIGHT; <depth> is not a multiple of four or equal to TEXTURE_DEPTH; <xoffset> <yoffset>, or <zoffset> is not a multiple of four.

New State

In table 6.17, Textures (page 278), increment the 42 in "n x Z42*" by 16 (or 12 if EXT_texture_compression_s3tc is not supported).

[NOTE: The OpenGL 2.0 specification actually should read "n x Z48*" because of the 6 generic compressed internal formats in table 3.18.]

New Implementation Dependent State

None

NVIDIA Implementation Details

GeForce FX, Quadro FX, and GeForce 6 and 7 Series GPUs store sRGB texels at 8 bits per component. sRGB conversion occurs post-filtering.

Revision History

0.8: Add issue 24 with recommendation for sRGB mipmap generation.

0.7: Add issue 23 about alternative implementation based on either GL_RGB12 or GL_RGB16 based on discussions with Jeremy Sandmel.

0.6: Add issue 22 about GL_COMPRESSED_TEXTURE_FORMATS.

0.5: Fix grammar, add issues 20 and 21 based on Brian Paul's feedback.

0.4: Update issue 18 based on Matrox feedback.

0.3: Update NV_texture_expand_normal interaction.

Name

    EXT_timer_query

Name Strings

    GL_EXT_timer_query

Contact

James Jones, NVIDIA Corporation (jajones 'at' nvidia.com)

Contributors

Axel Mamode, Sony
Brian Paul, Tungsten Graphics
Pat Brown, NVIDIA
Remi Arnaud, Sony

Status

Shipping (version 1.0)

Supported by NVIDIA Release 80 drivers.

Version

```
Last Modified Date:         11/6/2006
Revision:                   2
```

Number

319

Dependencies

Written based on the wording of the OpenGL 2.0 specification.

OpenGL 1.5 is required.

This extension modifies ARB_occlusion_query and NV_occlusion_query.

Overview

Applications can benefit from accurate timing information in a number of different ways. During application development, timing information can help identify application or driver bottlenecks. At run time, applications can use timing information to dynamically adjust the amount of detail in a scene to achieve constant frame rates. OpenGL implementations have historically provided little to no useful timing information. Applications can get some idea of timing by reading timers on the CPU, but these timers are not synchronized with the graphics rendering pipeline. Reading a CPU timer does not guarantee the completion of a potentially large amount of graphics work accumulated before the timer is read, and will thus produce wildly inaccurate results. glFinish() can be used to determine when previous rendering commands have been completed, but will idle the graphics pipeline and adversely affect application performance.

This extension provides a query mechanism that can be used to determine the amount of time it takes to fully complete a set of GL commands, and without stalling the rendering pipeline. It uses the query object mechanisms first introduced in the occlusion query extension, which allow time intervals to be polled asynchronously by the application.

Issues

What time interval is being measured?

RESOLVED: The timer starts when all commands prior to BeginQuery() have been fully executed. At that point, everything that should be drawn by those commands has been written to the framebuffer. The timer stops when all commands prior to EndQuery() have been fully executed.

What unit of time will time intervals be returned in?

RESOLVED: Nanoseconds (10^-9 seconds). This unit of measurement allows for reasonably accurate timing of even small blocks of rendering commands. The granularity of the timer is implementation-dependent. A 32-bit query counter can express intervals of up to approximately 4 seconds.

What should be the minimum number of counter bits for timer queries?

RESOLVED: 30 bits, which will allow timing sections that take up to 1 second to render.

How are counter results of more than 32 bits returned?

RESOLVED: Via two new datatypes, int64EXT and uint64EXT, and their corresponding GetQueryObject entry points. These types hold integer values and have a minimum bit width of 64.

Should the extension measure total time elapsed between the full completion of the BeginQuery and EndQuery commands, or just time spent in the graphics library?

RESOLVED: This extension will measure the total time elapsed between the full completion of these commands. Future extensions may implement a query to determine time elapsed at different stages of the graphics pipeline.

This extension introduces a second query type supported by BeginQuery/EndQuery. Can multiple query types be active simultaneously?

RESOLVED: Yes; an application may perform an occlusion query and a timer query simultaneously. An application can not perform multiple occlusion queries or multiple timer queries simultaneously. An application also can not use the same query object for an occlusion query and a timer query simultaneously.

Do query objects have a query type permanently associated with them?

RESOLVED: No. A single query object can be used to perform different types of queries, but not at the same time.

Having a fixed type for each query object simplifies some aspects of the implementation -- not having to deal with queries with different result sizes, for example. It would also mean that BeginQuery() with a query object of the "wrong" type would result in an INVALID_OPERATION error.

How predictable/repeatable are the results returned by the timer query?

RESOLVED: In general, the amount of time needed to render the same primitives should be fairly constant. But there may be many other system issues (e.g., context switching on the CPU and GPU, virtual memory page faults, memory cache behavior on the CPU and GPU) that can cause times to vary wildly.

Note that modern GPUs are generally highly pipelined, and may be processing different primitives in different pipeline stages simultaneously. In this extension, the timers start and stop when the BeginQuery/EndQuery commands reach the bottom of the rendering pipeline. What that means is that by the time the timer starts, the GL driver on the CPU may have started work on GL commands issued after BeginQuery, and the higher pipeline stages (e.g., vertex transformation) may have started as well.

What should the new 64 bit integer type be called?

RESOLVED: The new types will be called GLint64EXT/GLuint64EXT The new command suffixes will be i64 and ui64. These names clearly convey the minimum size of the types. These types are similar to the C99 standard type int_least64_t, but we use names similar to the C99 optional type int64_t for simplicity.

New Procedures and Functions

```
void GetQueryObjecti64vEXT(uint id, enum pname, int64EXT *params);
void GetQueryObjectui64vEXT(uint id, enum pname, uint64EXT *params);
```

New Tokens

Accepted by the <target> parameter of BeginQuery, EndQuery, and GetQueryiv:

```
    TIME_ELAPSED_EXT                               0x88BF
```

Additions to Chapter 2 of the OpenGL 2.0 Specification (OpenGL Operation)

(Modify table 2.1, Correspondence of command suffix letters to GL argument types, p. 8) Add two new types and suffixes:

```
Letter Corresponding GL Type
------ ---------------------
i64    int64EXT
ui64   uint64EXT
```

(Modify table 2.2, GL data types, p. 9) Add two new types:

GL Type	Minimum Bit Width	Description
int64EXT	64	signed 2's complement binary integer
uint64EXT	64	unsigned binary integer

Additions to Chapter 3 of the OpenGL 2.0 Specification (Rasterization)

None.

Additions to Chapter 4 of the OpenGL 2.0 Specification (Per-Fragment Operations and the Framebuffer)

(Replace Section 4.1.7, Occlusion Queries, p.204)

Section 4.1.7, Asynchronous Queries

Asynchronous queries provide a mechanism to return information about the processing of a sequence of GL commands. There are two query types supported by the GL. Occlusion queries (section 4.1.7.1) count the number of fragments or samples that pass the depth test. Timer queries (section 4.1.12) record the amount of time needed to fully process these commands.

The results of asynchronous queries are not returned by the GL immediately after the completion of the last command in the set; subsequent commands can be processed while the query results are not complete. When available, the query results are stored in an associated query object. The commands described in section 6.1.12 provide mechanisms to determine when query results are available and return the actual results of the query. The name space for query objects is the unsigned integers, with zero reserved by the GL.

Each type of query supported by the GL has an active query object name. If the active query object name for a query type is non-zero, the GL is currently tracking the information corresponding to that query type and the query results will be written into the corresponding query object. If the active query object for a query type name is zero, no such information is being tracked.

A query object is created by calling

```
    void BeginQuery(enum target, uint id);
```

with an unused name <id>. <target> indicates the type of query to be performed; valid values of <target> are defined in subsequent sections. When a query object is created, the name <id> is marked as used and associated with a new query object.

BeginQuery sets the active query object name for the query type given by <target> to <id>. If BeginQuery is called with an <id> of zero, if the active query object name for <target> is non-zero, or if <id> is the active query object name for any query type, the error INVALID_OPERATION is generated.

The command

```
    void EndQuery(enum target);
```

marks the end of the sequence of commands to be tracked for the query type given by <target>. The active query object for <target> is updated to indicate that query results are not available, and the active query object name for <target> is reset to zero. When the commands issued prior to EndQuery have completed and a final query result is available, the query object active when EndQuery is called is updated by the GL. The query object is updated to indicate that the query results are available and to contain the query result. If the active query object name for <target> is zero when EndQuery is called, the error INVALID_OPERATION is generated.

The command

```
    void GenQueries(sizei n, uint *ids);
```

returns <n> previously unused query object names in <ids>. These names are marked as used, but no object is associated with them until the first time they are used by BeginQuery.

Query objects are deleted by calling

```
    void DeleteQueries(sizei n, const uint *ids);
```

<ids> contains <n> names of query objects to be deleted. After a query object is deleted, its name is again unused. Unused names in <ids> are silently ignored.

Calling either GenQueries or DeleteQueries while any query of any target is active causes an INVALID_OPERATION error to be generated.

Query objects contain two pieces of state: a single bit indicating whether a query result is available, and an integer containing the query result value. The number of bits used to represent the query result is implementation-dependent. In the initial state of a query object, the result is available and its value is zero.

The necessary state for each query type is an unsigned integer holding the active query object name (zero if no query object is active), and any state necessary to keep the current results of an asynchronous query in progress.

Section 4.1.7.1, Occlusion Queries

Occlusion queries use query objects to track the number of fragments or samples that pass the depth test. An occlusion query can be started and finished by calling BeginQuery and EndQuery, respectively, with a <target> of SAMPLES_PASSED.

When an occlusion query starts, the samples-passed count maintained by the GL is set to zero. When an occlusion query is active, the samples-passed count is incremented for each fragment that passes the depth test. If the value of SAMPLE BUFFERS is 0, then the samples-passed count is incremented by 1 for each fragment. If the value of SAMPLE BUFFERS is 1, then the samples-passed count is incremented by the number of samples whose coverage bit is set. However, implementations, at their discretion, may instead increase the samples-passed count by the value of SAMPLES if any sample in the fragment is covered. When an occlusion query finishes and all fragments generated by the commands issued prior to EndQuery have been generated, the samples-passed count is written to the corresponding query object as the query result value, and the query result for that object is marked as available.

If the samples-passed count overflows, (i.e., exceeds the value 2^n - 1, where n is the number of bits in the samples-passed count), its value becomes undefined. It is recommended, but not required, that implementations handle this overflow case by saturating at 2^n - 1 and incrementing no further.

(Add new Section 4.1.12, Timer Queries, p.212)

Timer queries use query objects (section 4.1.7) to track the amount of time needed to fully complete a set of GL commands. A timer query can be started and finished by calling BeginQuery and EndQuery, respectively, with a <target> of TIME_ELAPSED_EXT.

When BeginQuery and EndQuery are called with a <target> of TIME_ELAPSED_EXT, the GL prepares to start and stop the timer used for timer queries. The timer is started or stopped when the effects from all previous commands on the GL client and server state and the framebuffer have been fully realized. The BeginQuery and EndQuery commands may return before the timer is actually started or stopped. When the timer query timer is finally stopped, the elapsed time (in nanoseconds) is written to the corresponding query object as the query result value, and the query result for that object is marked as available.

If the elapsed time overflows the number of bits, <n>, available to hold elapsed time, its value becomes undefined. It is recommended, but not required, that implementations handle this overflow case by saturating at 2^n - 1.

Additions to Chapter 5 of the OpenGL 2.0 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 2.0 Specification (State and State Requests)

(Replace Section 6.1.12, Occlusion Queries, p. 254)

Section 6.1.12, Asynchronous Queries

The command

```
  boolean IsQuery(uint id);
```

returns TRUE if <id> is the name of a query object. If <id> is zero, or if <id> is a non-zero value that is not the name of a query object, IsQuery returns FALSE.

Information about a query target can be queried with the command

```
  void GetQueryiv(enum target, enum pname, int *params);
```

<target> identifies the query target and can be SAMPLES_PASSED for occlusion queries or TIME_ELAPSED_EXT for timer queries.

If <pname> is CURRENT_QUERY, the name of the currently active query for <target>, or zero if no query is active, will be placed in <params>.

If <pname> is QUERY_COUNTER_BITS, the implementation-dependent number of bits used to hold the query result for <target> will be placed in params. The number of query counter bits may be zero, in which case the counter contains no useful information.

For occlusion queries (SAMPLES_PASSED), if the number of bits is non-zero, the minimum number of bits allowed is a function of the implementation's maximum viewport dimensions (MAX_VIEWPORT_DIMS). The counter must be able to represent at least two overdraws for every pixel in the viewport. The formula to compute the allowable minimum value (where n is the minimum number of bits) is:

```
  n = min(32, ceil(log_2(maxViewportWidth * maxViewportHeight * 2))).
```

For timer queries (TIME_ELAPSED_EXT), if the minimum number if bits is non-zero, it must be at least 30.

The state of a query object can be queried with the commands

```
  void GetQueryObjectiv(uint id, enum pname, int *params);
  void GetQueryObjectuiv(uint id, enum pname, uint *params);
  void GetQueryObjecti64vEXT(uint id, enum pname, int64EXT *params);
  void GetQueryObjectui64vEXT(uint id, enum pname, uint64EXT *params);
```
If <id> is not the name of a query object, or if the query object named by <id> is currently active, then an INVALID_OPERATION error is generated.

If <pname> is QUERY_RESULT, then the query object's result value is returned as a single integer in <params>. If the value is so large in magnitude that it cannot be represented with the requested type, then the nearest value representable using the requested type is returned. If the number of query counter bits for any <target> is zero, then the result is returned as a single integer with a value of 0.

There may be an indeterminate delay before the above query returns. If <pname> is QUERY_RESULT_AVAILABLE, FALSE is returned if such a delay would be required, TRUE is returned otherwise. It must always be true that if any query object returns a result available of TRUE, all queries of the same type issued prior to that query must also return TRUE.

Querying the state for any given query object forces the corresponding query to complete within a finite amount of time.

If multiple queries are issued using the same object name prior to calling GetQueryObject[u]iv, the result and availability information returned will always be from the last query issued. The results from any queries before the last one will be lost if they are not retrieved before starting a new query on the same <target> and <id>.

GLX Protocol (Modification to the GLX 1.3 Protocol Encoding Specification)

```
    Add to Section 1.4 (p.2), Common Types

        INT64       A 64-bit signed integer value.

        CARD64      A 64-bit unsigned integer value.

    Two new non-rendering GL commands are added.  These commands are sent
    seperately (i.e., not as part of a glXRender or glXRenderLarge request),
    using the glXVendorPrivateWithReply request:

        GetQueryObjecti64vEXT
            1           CARD8           opcode (X assigned)
            1           1328            GLX opcode (glXVendorPrivateWithReply)
            2           4               request length
            4           GLX_CONTEXT_TAG context tag
            4           CARD32          id
            4           ENUM            pname
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           m               reply length, m=(n==1?0:n)
            4                           unused
            4           CARD32          n

            if (n=1) this follows:

            8           INT64           params
            8                           unused

            otherwise this follows:

            16                          unused
            n*8         LISTofINT64     params
        GetQueryObjectui64vEXT
            1           CARD8           opcode (X assigned)
            1           1329            GLX opcode (glXVendorPrivateWithReply)
            2           4               request length
            4           GLX_CONTEXT_TAG context tag
            4           CARD32          id
            4           ENUM            pname
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           m               reply length, m=(n==1?0:n)
            4                           unused
            4           CARD32          n

           if (n=1) this follows:

            8           CARD64          params
            8                           unused

            otherwise this follows:

            16                          unused
            n*8         CARD64          params
```

Errors

All existing errors for query objects apply unchanged from the ARB_occlusion_query spec, except the modification below:

The error INVALID_ENUM is generated if BeginQueryARB, EndQueryARB, or GetQueryivARB is called where <target> is not SAMPLES_PASSED or TIME_ELAPSED_EXT.

The error INVALID_OPERATION is generated if GetQueryObjecti64vEXT or GetQueryObjectui64vEXT is called where <id> is not the name of a query object.

The error INVALID_OPERATION is generated if GetQueryObjecti64vEXT or GetQueryObjectui64vEXT is called where <id> is the name of a currently active query object.

The error INVALID_ENUM is generated if GetQueryObjecti64vEXT or GetQueryObjectui64vEXT is called where <pname> is not QUERY_RESULT or QUERY_RESULT_AVAILABLE.

New State

(table 6.37, p 298) Update the occlusion query / query object state to cover timer queries:

Get Value	Type	Get Command	Init. Value	Description	Sec	Attribute
CURRENT_QUERY	2xZ+	GetQueryiv	0	Active query object name (occlusion and timer)	4.1.7	-
QUERY_RESULT	2xZ+	GetQueryObjectiv	0	Query object result (samples passed or time elapsed)	4.1.7	-
QUERY_RESULT_AVAILABLE	2xB	GetQueryObjectiv	TRUE	Query object result available?	4.1.7	-

New Implementation Dependent State

(table 6.34, p. 295) Update the occlusion query / query object state to cover timer queries:

Get Value	Type	Get Command	Minimum Value	Description	Sec	Attribute
QUERY_COUNTER_BITS	2xZ+	GetQueryiv	see 6.1.12	Asynchronous query counter bits (occlusion and timer queries)	6.1.12	-

Dependencies on ARB_occlusion_query and NV_occlusion_query

If ARB_occlusion_query or NV_occlusion_query is supported, the previous spec edits are considered to apply to the nearly identical language in these extension specifications. Note that the functionality provided by these extensions is included in OpenGL versions 1.5 and greater.

Usage Examples

Here is some rough sample code that demonstrates the intended usage of this extension.

```
GLint queries[N];
GLint available = 0;
// timer queries can contain more than 32 bits of data, so always
// query them using the 64 bit types to avoid overflow
GLuint64EXT timeElapsed = 0;

// Create a query object.
glGenQueries(N, queries);

// Start query 1
glBeginQuery(GL_TIME_ELAPSED_EXT, queries[0]);

// Draw object 1
....

// End query 1
glEndQuery(GL_TIME_ELAPSED_EXT);

...

// Start query N
glBeginQuery(GL_TIME_ELAPSED_EXT, queries[N-1]);

// Draw object N
....

// End query N
glEndQuery(GL_TIME_ELAPSED_EXT);

// Wait for all results to become available
while (!available) {
    glGetQueryObjectiv(queries[N-1], GL_QUERY_RESULT_AVAILABLE, &available);
}

for (i = 0; i < N; i++) {
    // See how much time the rendering of object i took in nanoseconds.
    glGetQueryObjectui64vEXT(queries[i], GL_QUERY_RESULT, &timeElapsed);

    // Do something useful with the time.  Note that care should be
    // taken to use all significant bits of the result, not just the
    // least significant 32 bits.
    AdjustObjectLODBasedOnDrawTime(i, timeElapsed);
}
```

This example is sub-optimal in that it stalls at the end of every frame to wait for query results. Ideally, the collection of results would be delayed one frame to minimize the amount of time spent waiting for the GPU to finish rendering.

Revision History

none yet

Name

    EXT_vertex_array

Name Strings

    GL_EXT_vertex_array

Version

    $Date: 1995/10/03 05:39:58 $ $Revision: 1.16 $ FINAL

Number

    30

Dependencies

    None

Overview

This extension adds the ability to specify multiple geometric primitives with very few subroutine calls. Instead of calling an OpenGL procedure to pass each individual vertex, normal, or color, separate arrays of vertexes, normals, and colors are prespecified, and are used to define a sequence of primitives (all of the same type) when a single call is made to DrawArraysEXT. A stride mechanism is provided so that an application can choose to keep all vertex data staggered in a single array, or sparsely in separate arrays. Single-array storage may optimize performance on some implementations.

This extension also supports the rendering of individual array elements, each specified as an index into the enabled arrays.

Issues

*   Should arrays for material parameters be provided? If so, how?

    A: No. Let's leave this to a separate extension, and keep this extension lean.

*   Should a FORTRAN interface be specified in this document?

*   It may not be possible to implement GetPointervEXT in FORTRAN. If not, should we eliminate it from this proposal?

    A: Leave it in.

*   Should a stride be specified by DrawArraysEXT which, if non-zero, would override the strides specified for the individual arrays? This might improve the efficiency of single-array transfers.

    A: No, it's not worth the effort and complexity.

```
    *   Should entry points for byte vertexes, byte indexes, and byte
        texture coordinates be added in this extension?

        A: No, do this in a separate extension, which defines byte support
           for arrays and for the current procedural interface.

    *   Should support for meshes (not strips) of rectangles be provided?

        A: No. If this is necessary, define a separate quad_mesh extension
           that supports both immediate mode and arrays.  (Add QUAD_MESH_EXT
           as an token accepted by Begin and DrawArraysEXT.  Add
           QuadMeshLengthEXT to specify the length of the mesh.)
```

Reasoning

```
    *   DrawArraysEXT requires that VERTEX_ARRAY_EXT be enabled so that
        future extensions can support evaluation as well as direct
        specification of vertex coordinates.

    *   This extension does not support evaluation.  It could be extended
        to provide such support by adding arrays of points to be evaluated,
        and by adding enables to indicate that the arrays are to be
        evaluated.  I think we may choose to add an array version of
        EvalMesh, rather than extending the operation of DrawArraysEXT,
        so I'd rather wait on this one.

    *   <size> is specified before <type> to match the order of the
        information in immediate mode commands, such as Vertex3f.
        (first 3, then f)

    *   It seems reasonable to allow attribute values to be undefined after
        DrawArraysEXT executes.  This avoids implementation overhead in
        the case where an incomplete primitive is specified, and will allow
        optimization on multiprocessor systems.  I don't expect this to be
        a burden to programmers.

    *   It is not an error to call VertexPointerEXT, NormalPointerEXT,
        ColorPointerEXT, IndexPointerEXT, TexCoordPointerEXT,
        or EdgeFlagPointerEXT between the execution of Begin and the
        corresponding execution of End.  Because these commands will
        typically be implemented on the client side with no protocol,
        testing for between-Begin-End status requires that the client
        track this state, or that a round trip be made.  Neither is
        desirable.

    *   Arrays are enabled and disabled individually, rather than with a
        single mask parameter, for two reasons.  First, we have had trouble
        allocating bits in masks, so eliminating a mask eliminates potential
        trouble down the road.  We may eventually require a larger number of
        array types than there are bits in a mask.  Second, making the
        enables into state eliminates a parameter in ArrayElementEXT, and
        may allow it to execute more efficiently.  Of course this state
        model may result in programming errors, but OpenGL is full of such
        hazards anyway!

    *   ArrayElementEXT is provided to support applications that construct
        primitives by indexing vertex data, rather than by streaming through
```

arrays of data in first-to-last order. Because each call specifies only a single vertex, it is possible for an application to explicitly specify per-primitive attributes, such as a single normal per individual triangle.

* The <count> parameters are added to the *PointerEXT commands to allow implementations to cache array data, and in particular to cache the transformed results of array data that are rendered repeatedly by ArrayElementEXT. Implementations that do not wish to perform such caching can ignore the <count> parameter.

* The <first> parameter of DrawArraysEXT allows a single set of arrays to be used repeatedly, possibly improving performance.

New Procedures and Functions

```
    void ArrayElementEXT(int i);

    void DrawArraysEXT(enum mode,
                       int first,
                       sizei count);

    void VertexPointerEXT(int size,
                          enum type,
                          sizei stride,
                          sizei count,
                          const void* pointer);

    void NormalPointerEXT(enum type,
                          sizei stride,
                          sizei count,
                          const void* pointer);

    void ColorPointerEXT(int size,
                         enum type,
                         sizei stride,
                         sizei count,
                         const void* pointer);

    void IndexPointerEXT(enum type,
                         sizei stride,
                         sizei count,
                         const void* pointer);

    void TexCoordPointerEXT(int size,
                            enum type,
                            sizei stride,
                            sizei count,
                            const void* pointer);

    void EdgeFlagPointerEXT(sizei stride,
                            sizei count,
                            const Boolean* pointer);

    void GetPointervEXT(enum pname,
                       void** params);
```

New Tokens

```
    Accepted by the <cap> parameter of Enable, Disable, and IsEnabled, and
    by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and
    GetDoublev:

        VERTEX_ARRAY_EXT                0x8074
        NORMAL_ARRAY_EXT                0x8075
        COLOR_ARRAY_EXT                 0x8076
        INDEX_ARRAY_EXT                 0x8077
        TEXTURE_COORD_ARRAY_EXT         0x8078
        EDGE_FLAG_ARRAY_EXT             0x8079

    Accepted by the <type> parameter of VertexPointerEXT, NormalPointerEXT,
    ColorPointerEXT, IndexPointerEXT, and TexCoordPointerEXT:

        DOUBLE_EXT                      0x140A

    Accepted by the <pname> parameter of GetBooleanv, GetIntegerv,
    GetFloatv, and GetDoublev:

        VERTEX_ARRAY_SIZE_EXT           0x807A
        VERTEX_ARRAY_TYPE_EXT           0x807B
        VERTEX_ARRAY_STRIDE_EXT         0x807C
        VERTEX_ARRAY_COUNT_EXT          0x807D
        NORMAL_ARRAY_TYPE_EXT           0x807E
        NORMAL_ARRAY_STRIDE_EXT         0x807F
        NORMAL_ARRAY_COUNT_EXT          0x8080
        COLOR_ARRAY_SIZE_EXT            0x8081
        COLOR_ARRAY_TYPE_EXT            0x8082
        COLOR_ARRAY_STRIDE_EXT          0x8083
        COLOR_ARRAY_COUNT_EXT           0x8084
        INDEX_ARRAY_TYPE_EXT            0x8085
        INDEX_ARRAY_STRIDE_EXT          0x8086
        INDEX_ARRAY_COUNT_EXT           0x8087
        TEXTURE_COORD_ARRAY_SIZE_EXT    0x8088
        TEXTURE_COORD_ARRAY_TYPE_EXT    0x8089
        TEXTURE_COORD_ARRAY_STRIDE_EXT  0x808A
        TEXTURE_COORD_ARRAY_COUNT_EXT   0x808B
        EDGE_FLAG_ARRAY_STRIDE_EXT      0x808C
        EDGE_FLAG_ARRAY_COUNT_EXT       0x808D

    Accepted by the <pname> parameter of GetPointervEXT:

        VERTEX_ARRAY_POINTER_EXT        0x808E
        NORMAL_ARRAY_POINTER_EXT        0x808F
        COLOR_ARRAY_POINTER_EXT         0x8090
        INDEX_ARRAY_POINTER_EXT         0x8091
        TEXTURE_COORD_ARRAY_POINTER_EXT 0x8092
        EDGE_FLAG_ARRAY_POINTER_EXT     0x8093
```

Additions to Chapter 2 of the 1.0 Specification (OpenGL Operation)

Array Specification

Individual array pointers and associated data are maintained for an
array of vertexes, an array of normals, an array of colors, an array

of color indexes, an array of texture coordinates, and an array of edge flags. The data associated with each array specify the data type of the values in the array, the number of values per element in the array (e.g. vertexes of 2, 3, or 4 coordinates), the byte stride from one array element to the next, and the number of elements (counting from the first) that are static. Static elements may be modified by the application, but once they are modified, the application must explicitly respecify the array before using it for any rendering. When an array is specified, the pointer and associated data are saved as client-side state, and static elements may be cached by the implementation. Non-static (dynamic) elements are never accessed until ArrayElementEXT or DrawArraysEXT is issued.

VertexPointerEXT specifies the location and data format of an array of vertex coordinates. <pointer> specifies a pointer to the first coordinate of the first vertex in the array. <type> specifies the data type of each coordinate in the array, and must be one of SHORT, INT, FLOAT, or DOUBLE_EXT, implying GL data types short, int, float, and double respectively. <size> specifies the number of coordinates per vertex, and must be 2, 3, or 4. <stride> specifies the byte offset between pointers to consecutive vertexes. If <stride> is zero, the vertex data are understood to be tightly packed in the array. <count> specifies the number of vertexes, counting from the first, that are static.

NormalPointerEXT specifies the location and data format of an array of normals. <pointer> specifies a pointer to the first coordinate of the first normal in the array. <type> specifies the data type of each coordinate in the array, and must be one of BYTE, SHORT, INT, FLOAT, or DOUBLE_EXT, implying GL data types byte, short, int, float, and double respectively. It is understood that each normal comprises three coordinates. <stride> specifies the byte offset between pointers to consecutive normals. If <stride> is zero, the normal data are understood to be tightly packed in the array. <count> specifies the number of normals, counting from the first, that are static.

ColorPointerEXT specifies the location and data format of an array of color components. <pointer> specifies a pointer to the first component of the first color element in the array. <type> specifies the data type of each component in the array, and must be one of BYTE, UNSIGNED_BYTE, SHORT, UNSIGNED_SHORT, INT, UNSIGNED_INT, FLOAT, or DOUBLE_EXT, implying GL data types byte, ubyte, short, ushort, int, uint, float, and double respectively. <size> specifies the number of components per color, and must be 3 or 4. <stride> specifies the byte offset between pointers to consecutive colors. If <stride> is zero, the color data are understood to be tightly packed in the array. <count> specifies the number of colors, counting from the first, that are static.

IndexPointerEXT specifies the location and data format of an array of color indexes. <pointer> specifies a pointer to the first index in the array. <type> specifies the data type of each index in the array, and must be one of SHORT, INT, FLOAT, or DOUBLE_EXT, implying GL data types short, int, float, and double respectively. <stride> specifies the byte offset between pointers to consecutive indexes. If <stride> is zero, the index data are understood to be tightly packed in the array. <count> specifies the number of indexes, counting from the first, that are static.

TexCoordPointerEXT specifies the location and data format of an array of texture coordinates. <pointer> specifies a pointer to the first coordinate of the first element in the array. <type> specifies the data type of each coordinate in the array, and must be one of SHORT, INT, FLOAT, or DOUBLE_EXT, implying GL data types short, int, float, and double respectively. <size> specifies the number of coordinates per element, and must be 1, 2, 3, or 4. <stride> specifies the byte offset between pointers to consecutive elements of coordinates. If <stride> is zero, the coordinate data are understood to be tightly packed in the array. <count> specifies the number of texture coordinate elements, counting from the first, that are static.

EdgeFlagPointerEXT specifies the location and data format of an array of boolean edge flags. <pointer> specifies a pointer to the first flag in the array. <stride> specifies the byte offset between pointers to consecutive edge flags. If <stride> is zero, the edge flag data are understood to be tightly packed in the array. <count> specifies the number of edge flags, counting from the first, that are static.

The table below summarizes the sizes and data types accepted (or understood implicitly) by each of the six pointer-specification commands.

Command	Sizes	Types
VertexPointerEXT	2,3,4	short, int, float, double
NormalPointerEXT	3	byte, short, int, float, double
ColorPointerEXT	3,4	byte, short, int, float, double, ubyte, ushort, uint
IndexPointerEXT	1	short, int, float, double
TexCoordPointerEXT	1,2,3,4	short, int, float, double
EdgeFlagPointerEXT	1	boolean

Rendering the Arrays

By default all the arrays are disabled, meaning that they will not be accessed when either ArrayElementEXT or DrawArraysEXT is called. An individual array is enabled or disabled by calling Enable or Disable with <cap> set to appropriate value, as specified in the table below:

Array Specification Command	Enable Token
VertexPointerEXT	VERTEX_ARRAY_EXT
NormalPointerEXT	NORMAL_ARRAY_EXT
ColorPointerEXT	COLOR_ARRAY_EXT
IndexPointerEXT	INDEX_ARRAY_EXT
TexCoordPointerEXT	TEXTURE_COORD_ARRAY_EXT
EdgeFlagPointerEXT	EDGE_FLAG_ARRAY_EXT

When ArrayElementEXT is called, a single vertex is drawn, using vertex and attribute data taken from location <i> of the enabled arrays. The semantics of ArrayElementEXT are defined in the C-code below:

```
        void ArrayElementEXT (int i) {
            byte* p;
            if (NORMAL_ARRAY_EXT) {
                if (normal_stride == 0)
                    p = (byte*)normal_pointer + i * 3 * sizeof(normal_type);
                else
                    p = (byte*)normal_pointer + i * normal_stride;
                Normal3<normal_type>v ((normal_type*)p);
            }
            if (COLOR_ARRAY_EXT) {
                if (color_stride == 0)
                    p = (byte*)color_pointer +
                        i * color_size * sizeof(color_type);
                else
                    p = (byte*)color_pointer + i * color_stride;
                Color<color_size><color_type>v ((color_type*)p);
            }
            if (INDEX_ARRAY_EXT) {
                if (index_stride == 0)
                    p = (byte*)index_pointer + i * sizeof(index_type);
                else
                    p = (byte*)index_pointer + i * index_stride;
                Index<index_type>v ((index_type*)p);
            }
            if (TEXTURE_COORD_ARRAY_EXT) {
                if (texcoord_stride == 0)
                    p = (byte*)texcoord_pointer +
                        i * texcoord_size * sizeof(texcoord_type);
                else
                    p = (byte*)texcoord_pointer + i * texcoord_stride;
                TexCoord<texcoord_size><texcoord_type>v ((texcoord_type*)p);
            }
            if (EDGE_FLAG_ARRAY_EXT) {
                if (edgeflag_stride == 0)
                    p = (byte*)edgeflag_pointer + i * sizeof(boolean);
                else
                    p = (byte*)edgeflag_pointer + i * edgeflag_stride;
                EdgeFlagv ((boolean*)p);
            }
            if (VERTEX_ARRAY_EXT) {
                if (vertex_stride == 0)
                    p = (byte*)vertex_pointer +
                        i * vertex_size * sizeof(vertex_type);
                else
                    p = (byte*)vertex_pointer + i * vertex_stride;
                Vertex<vertex_size><vertex_type>v ((vertex_type*)p);
            }
        }
```

ArrayElementEXT executes even if VERTEX_ARRAY_EXT is not enabled. No drawing occurs in this case, but the attributes corresponding to enabled arrays are modified.

When DrawArraysEXT is called, <count> sequential elements from each enabled array are used to construct a sequence of geometric primitives, beginning with element <first>. <mode> specifies what kind of primitives are constructed, and how the array elements are used to construct these primitives. Accepted values for <mode> are POINTS, LINE_STRIP, LINE_LOOP, LINES, TRIANGLE_STRIP, TRIANGLE_FAN, TRIANGLES, QUAD_STRIP, QUADS, and POLYGON. If VERTEX_ARRAY_EXT is not enabled, no geometric primitives are generated.

The semantics of DrawArraysEXT are defined in the C-code below:

```
void DrawArraysEXT(enum mode, int first, sizei count) {
    int i;
    if (count < 0)
        /* generate INVALID_VALUE error and abort */
    else {
        Begin (mode);
        for (i=0; i < count; i++)
            ArrayElementEXT(first + i);
        End ();
    }
}
```

The ways in which the execution of DrawArraysEXT differs from the semantics indicated in the pseudo-code above are:

1. Vertex attributes that are modified by DrawArraysEXT have an unspecified value after DrawArraysEXT returns. For example, if COLOR_ARRAY_EXT is enabled, the value of the current color is undefined after DrawArraysEXT executes. Attributes that aren't modified remain well defined.

2. Operation of DrawArraysEXT is atomic with respect to error generation. If an error is generated, no other operations take place.

Although it is not an error to respecify an array between the execution of Begin and the corresponding execution of End, the result of such respecification is undefined. Static array data may be read and cached by the implementation at any time. If static array data are modified by the application, the results of any subsequently issued ArrayElementEXT or DrawArraysEXT commands are undefined.

Additions to Chapter 3 of the 1.0 Specification (Rasterization)

None

Additions to Chapter 4 of the 1.0 Specification (Per-Fragment Operations and the Frame buffer)

None

Additions to Chapter 5 of the 1.0 Specification (Special Functions)

ArrayElementEXT and DrawArraysEXT are included in display lists. When either command is entered into a display list, the necessary array data (determined by the array pointers and enables) is also entered into the display list. Because the array pointers and enables are client side state, their values affect display lists when the lists are created, not when the lists are executed.

Array specification commands VertexPointerEXT, NormalPointerEXT, ColorPointerEXT, IndexPointerEXT, TexCoordPointerEXT, and EdgeFlagPointerEXT specify client side state, and are therefore not included in display lists. Likewise Enable and Disable, when called with <cap> set to VERTEX_ARRAY_EXT, NORMAL_ARRAY_EXT, COLOR_ARRAY_EXT, INDEX_ARRAY_EXT, TEXTURE_COORD_ARRAY_EXT, or EDGE_FLAG_ARRAY_EXT, are not included in display lists. GetPointervEXT returns state information, and so is not included in display lists.

Additions to Chapter 6 of the 1.0 Specification (State and State Requests)

GetPointervEXT returns in <param> the array pointer value specified by <pname>. Accepted values for <pname> are VERTEX_ARRAY_POINTER_EXT, NORMAL_ARRAY_POINTER_EXT, COLOR_ARRAY_POINTER_EXT, INDEX_ARRAY_POINTER_EXT, TEXTURE_COORD_ARRAY_POINTER_EXT, and EDGE_FLAG_ARRAY_POINTER_EXT.

All array data are client side state, and are not saved or restored by PushAttrib and PopAttrib.

Additions to the GLX Specification

None

GLX Protocol

A new rendering command is added; it can be sent to the server as part of a glXRender request or as part of a glXRenderLarge request:

The DrawArraysEXT command consists of three sections, in the following order: (1) header information, (2) a list of array information, containing the type and size of the array values for each enabled array and (3) a list of vertex data. Each element in the list of vertex data contains information for a single vertex taken from the enabled arrays.

```
DrawArraysEXT
    2           16+(12*m)+(s*n) rendering command length
    2           4116            rendering command opcode
    4           CARD32          n (number of array elements)
    4           CARD32          m (number of enabled arrays)
    4           ENUM            mode    /* GL_POINTS etc */
    12*m        LISTofARRAY_INFO
    s*n         LISTofVERTEX_DATA
```

Where s = ns + cs + is + ts + es + vs + np + cp + ip + tp + ep + vp. (See description below, under VERTEX_DATA.) Note that if an array is disabled then no information is sent for it. For example, when the normal array is disabled, there is no ARRAY_INFO record for the normal array and ns and np are both zero.

Note that the list of ARRAY_INFO is unordered: since the ARRAY_INFO record contains the array type, the arrays in the list may be stored in any order. Also, the VERTEX_DATA list is a packed list of vertices. For each vertex, data is retrieved from the enabled arrays, and stored in the list.

```
        If the command is encoded in a glXRenderLarge request, the command
        opcode and command length fields above are expanded to 4 bytes each:

        4           20+(12*m)+(s*n) rendering command length
        4           4116            rendering command opcode

    ARRAY_INFO
        4           ENUM                    data type
                    0x1400  i=1             BYTE
                    0x1401  i=1             UNSIGNED_BYTE
                    0x1402  i=2             SHORT
                    0x1403  i=2             UNSIGNED_SHORT
                    0x1404  i=4             INT
                    0x1405  i=4             UNSIGNED_INT
                    0x1406  i=4             FLOAT
                    0x140A  i=8             DOUBLE_EXT
        4           INT32                   j (number of values in array element)
        4           ENUM                    array type
                    0x8074  j=2/3/4         VERTEX_ARRAY_EXT
                    0x8075  j=3             NORMAL_ARRAY_EXT
                    0x8076  j=3/4           COLOR_ARRAY_EXT
                    0x8077  j=1             INDEX_ARRAY_EXT
                    0x8078  j=1/2/3/4       TEXTURE_COORD_ARRAY_EXT
                    0x8079  j=1             EDGE_FLAG_ARRAY_EXT

        For each array, the size of an array element is i*j. Some arrays
        (e.g., the texture coordinate array) support different data sizes;
        for these arrays, the size, j, is specified when the array is defined.

    VERTEX_DATA
        if the normal array is enabled:

        ns          LISTofBYTE              normal array element
        np                                  unused, np=pad(ns)

        if the color array is enabled:

        cs          LISTofBYTE              color array element
        cp                                  unused, cp=pad(cs)

        if the index array is enabled:

        is          LISTofBYTE              index array element
        ip                                  unused, ip=pad(is)

        if the texture coord array is enabled:

        ts          LISTofBYTE              texture coord array element
        tp                                  unused, tp=pad(ts)

        if the edge flag array is enabled:

        es          LISTofBYTE              edge flag array element
        ep                                  unused, ep=pad(es)
        if the vertex array is enabled:

        vs          LISTofBYTE              vertex array element
        vp                                  unused, vp=pad(vs)

        where ns, cs, is, ts, es, vs is the size of the normal, color, index,
        texture, edge and vertex array elements and np, cp, ip, tp, ep, vp is
        the padding for the normal, color, index, texture, edge and vertex array
        elements, respectively.
```

Errors

INVALID_OPERATION is generated if DrawArraysEXT is called between the execution of Begin and the corresponding execution of End.

INVALID_ENUM is generated if DrawArraysEXT parameter <mode> is not POINTS, LINE_STRIP, LINE_LOOP, LINES, TRIANGLE_STRIP, TRIANGLE_FAN, TRIANGLES, QUAD_STRIP, QUADS, or POLYGON.

INVALID_VALUE is generated if DrawArraysEXT parameter <count> is negative.

INVALID_VALUE is generated if VertexPointerEXT parameter <size> is not 2, 3, or 4.

INVALID_ENUM is generated if VertexPointerEXT parameter <type> is not SHORT, INT, FLOAT, or DOUBLE_EXT.

INVALID_VALUE is generated if VertexPointerEXT parameter <stride> or <count> is negative.

INVALID_ENUM is generated if NormalPointerEXT parameter <type> is not BYTE, SHORT, INT, FLOAT, or DOUBLE_EXT.

INVALID_VALUE is generated if NormalPointerEXT parameter <stride> or <count> is negative.

INVALID_VALUE is generated if ColorPointerEXT parameter <size> is not 3 or 4.

INVALID_ENUM is generated if ColorPointerEXT parameter <type> is not BYTE, UNSIGNED_BYTE, SHORT, UNSIGNED_SHORT, INT, UNSIGNED_INT, FLOAT, or DOUBLE_EXT.

INVALID_VALUE is generated if ColorPointerEXT parameter <stride> or <count> is negative.

INVALID_ENUM is generated if IndexPointerEXT parameter <type> is not SHORT, INT, FLOAT, or DOUBLE_EXT.

INVALID_VALUE is generated if IndexPointerEXT parameter <stride> or <count> is negative.

INVALID_VALUE is generated if TexCoordPointerEXT parameter <size> is not 1, 2, 3, or 4.

INVALID_ENUM is generated if TexCoordPointerEXT parameter <type> is not SHORT, INT, FLOAT, or DOUBLE_EXT.

INVALID_VALUE is generated if TexCoordPointerEXT parameter <stride> or <count> is negative.

INVALID_VALUE is generated if EdgeFlagPointerEXT parameter <stride> or <count> is negative.

INVALID_ENUM is generated if GetPointervEXT parameter <pname> is not VERTEX_ARRAY_POINTER_EXT, NORMAL_ARRAY_POINTER_EXT, COLOR_ARRAY_POINTER_EXT, INDEX_ARRAY_POINTER_EXT, TEXTURE_COORD_ARRAY_POINTER_EXT, or EDGE_FLAG_ARRAY_POINTER_EXT.

New State

Get Value	Get Command	Type	Initial Value	Attrib
VERTEX_ARRAY_EXT	IsEnabled	B	False	client
VERTEX_ARRAY_SIZE_EXT	GetIntegerv	Z+	4	client
VERTEX_ARRAY_TYPE_EXT	GetIntegerv	Z4	FLOAT	client
VERTEX_ARRAY_STRIDE_EXT	GetIntegerv	Z+	0	client
VERTEX_ARRAY_COUNT_EXT	GetIntegerv	Z+	0	client
VERTEX_ARRAY_POINTER_EXT	GetPointervEXT	Z+	0	client
NORMAL_ARRAY_EXT	IsEnabled	B	False	client
NORMAL_ARRAY_TYPE_EXT	GetIntegerv	Z5	FLOAT	client
NORMAL_ARRAY_STRIDE_EXT	GetIntegerv	Z+	0	client
NORMAL_ARRAY_COUNT_EXT	GetIntegerv	Z+	0	client
NORMAL_ARRAY_POINTER_EXT	GetPointervEXT	Z+	0	client
COLOR_ARRAY_EXT	IsEnabled	B	False	client
COLOR_ARRAY_SIZE_EXT	GetIntegerv	Z+	4	client
COLOR_ARRAY_TYPE_EXT	GetIntegerv	Z8	FLOAT	client
COLOR_ARRAY_STRIDE_EXT	GetIntegerv	Z+	0	client
COLOR_ARRAY_COUNT_EXT	GetIntegerv	Z+	0	client
COLOR_ARRAY_POINTER_EXT	GetPointervEXT	Z+	0	client
INDEX_ARRAY_EXT	IsEnabled	B	False	client
INDEX_ARRAY_TYPE_EXT	GetIntegerv	Z4	FLOAT	client
INDEX_ARRAY_STRIDE_EXT	GetIntegerv	Z+	0	client
INDEX_ARRAY_COUNT_EXT	GetIntegerv	Z+	0	client
INDEX_ARRAY_POINTER_EXT	GetPointervEXT	Z+	0	client
TEXTURE_COORD_ARRAY_EXT	IsEnabled	B	False	client
TEXTURE_COORD_ARRAY_SIZE_EXT	GetIntegerv	Z+	4	client
TEXTURE_COORD_ARRAY_TYPE_EXT	GetIntegerv	Z4	FLOAT	client
TEXTURE_COORD_ARRAY_STRIDE_EXT	GetIntegerv	Z+	0	client
TEXTURE_COORD_ARRAY_COUNT_EXT	GetIntegerv	Z+	0	client
TEXTURE_COORD_ARRAY_POINTER_EXT	GetPointervEXT	Z+	0	client
EDGE_FLAG_ARRAY_EXT	IsEnabled	B	False	client
EDGE_FLAG_ARRAY_STRIDE_EXT	GetIntegerv	Z+	0	client
EDGE_FLAG_ARRAY_COUNT_EXT	GetIntegerv	Z+	0	client
EDGE_FLAG_ARRAY_POINTER_EXT	GetPointervEXT	Z+	0	client

New Implementation Dependent State

None

Name

    EXT_vertex_weighting

Name Strings

    GL_EXT_vertex_weighting

Notice

Status

    Shipping (version 1.0)

Version

    NVIDIA Date: May 25, 2000

Number

    188

Dependencies

    None

    Written based on the wording of the OpenGL 1.2 specification but not
    dependent on it.

Overview

    The intent of this extension is to provide a means for blending
    geometry based on two slightly differing modelview matrices.
    The blending is based on a vertex weighting that can change on a
    per-vertex basis. This provides a primitive form of skinning.

    A second modelview matrix transform is introduced. When vertex
    weighting is enabled, the incoming vertex object coordinates are
    transformed by both the primary and secondary modelview matrices;
    likewise, the incoming normal coordinates are transformed by the
    inverses of both the primary and secondary modelview matrices.
    The resulting two position coordinates and two normal coordinates
    are blended based on the per-vertex vertex weight and then combined
    by addition. The transformed, weighted, and combined vertex position
    and normal are then used by OpenGL as the eye-space position and
    normal for lighting, texture coordinate, generation, clipping,
    and further vertex transformation.

Issues

    *Should the extension be written to extend to more than two vertex
    weights and modelview matrices?*

      RESOLUTION: NO. Supports only one vertex weight and two modelview
      matrices. If more than two is useful, that can be handled with

another extension.

Should the weighting factor be GLclampf instead of GLfloat?

RESOLUTION: GLfloat. Though the value of a weighting factors outside the range of zero to one (and even weights that do not add to one) is dubious, there is no reason to limit the implementation to values between zero and one.

Should the weights and modelview matrices be labeled 1 & 2 or 0 & 1?

RESOLUTION: 0 & 1. This is consistent with the way lights and texture units are named in OpenGL. Make GL_MODELVIEW0_EXT be an alias for GL_MODELVIEW. Note that the GL_MODELVIEW0_EXT+1 will not be GL_MODELVIEW1_EXT as is the case with GL_LIGHT0 and GL_LIGHT1.

Should there be a way to simultaneously Rotate, Translate, Scale, LoadMatrix, MultMatrix, etc. the two modelview matrices together?

RESOLUTION: NO. The application must use MatrixMode and repeated calls to keep the matrices in sync if desired.

Should the secondary modelview matrix stack be as deep as the primary matrix stack or can they be different sizes?

RESOLUTION: Must be the SAME size. This wastes a lot of memory that will be probably never be used (the modelview matrix stack must have at least 32 entries), but memory is cheap.

The value returned by MAX_MODELVIEW_STACK_DEPTH applies to both modelview matrices.

Should there be any vertex array support for vertex weights.

RESOLUTION: YES.

Should we have a VertexWeight2fEXT that takes has two weight values?

RESOLUTION: NO. The weights are always vw and 1-vw.

What is the "correct" way to blend matrices, particularly when wo is not one or the modelview matrix is projective?

RESOLUTION: While it may not be 100% correct, the extension blends the vertices based on transforming the object coordinates by both M0 and M1, but the resulting w coordinate comes from simply transforming the object coordinates by M0 and extracting the w.

Another option would be to simply blend the two sets of eye coordinates without any special handling of w. This is harder.

Another option would be to divide by w before blending the two sets of eye coordinates. This is awkward because if the weight is 1.0 with vertex weighting enabled, the result is not the same as disabling vertex weighting since EYE_LINEAR texgen is based of of the non-perspective corrected eye coordinates.

As specified, the normal weighting and combination is performed on unnormalized normals. Would the math work better if the normals were normalized before weighting and combining?

RESOLUTION: Vertex weighting of normals is after the GL_RESCALE_NORMAL step and before the GL_NORMALIZE step.

As specified, feedback and selection should apply vertex weighting if enabled. Yuck, that would mean that we need software code for vertex weighting.

RESOLUTION: YES, it should work with feedback and selection.

Sometimes it would be useful to mirror changes in both modelview matrices. For example, the viewing transforms are likely to be different, just the final modeling transforms would be different. Should there be an API support for mirroring transformations into both matrices?

RESOLUTION: NO. Such support is likely to complicate the matrix management in the OpenGL. Applications can do a Get matrix from modelview0 and then a LoadMatrix into modelview1 manually if they need to mirror things.

I also worry that if we had a mirrored matrix mode, it would double the transform concatenation work if used naively.

Many of the changes to the two modelview matrices will be the same. For example, the initial view transform loaded into each will be the same. Should there be a way to "mirror" changes to both modelview matrices?

RESOLUTION: NO. Mirroring matrix changes would complicate the driver's management of matrices. Also, I am worried that naive users would mirror all transforms and lead to lots of redundant matrix concatenations. The most efficient way to handle the slight differences between the modelview matrices is simply to GetFloat the primary matrix, LoadMatrix the values in the secondary modelview matrix, and then perform the "extra" transform to the secondary modelview matrix.

Ideally, a glCopyMatrix(GLenum src, GLenum dst) type OpenGL command could make this more efficient. There are similiar cases where you want the modelview matrix mirrored in the texture matrix. This is not the extension to solve this minor problem.

The post-vertex weighting normal is unlikely to be normalized. Should this extension automatically enable normalization?

RESOLUTION: NO. Normalization should operate as specified. The user is responsible for enabling GL_RESCALE_NORMAL or GL_NORMALIZE as needed.

You could imagine cases where the application only sent vertex weights of either zero or one and pre-normalized normals so that GL_NORMALIZE would not strictly be required.

Note that the vertex weighting of transformed normals occurs BEFORE normalize and AFTER rescaling. See the issue below for why this can make a difference.

How does vertex weighting interact with OpenGL 1.2's GL_RESCALE_NORMAL enable?

RESOLUTION: Vertex weighting of transformed normals occurs BEFORE normalize and AFTER rescaling.

OpenGL 1.2 permits normal rescaling to behave just like normalize and because normalize immediately follows rescaling, enabling rescaling can be implementied by simply always enabling normalize.

Vertex weighting changes this. If one or both of the modelview matrices has a non-uniform scale, it may be useful to enable rescaling and normalize and this operates differently than simply enabling normalize. The difference is that rescaling occurs before the normal vertex weighting.

An implementation that truly treated rescaling as a normalize would support both a pre-weighting normalize and a post-weighting normalize. Arguably, this is a good thing.

For implementations that perform simply rescaling and not a full normalize to implement rescaling, the rescaling factor can be concatenated into each particular inverse modelview matrix.

New Procedures and Functions

```
void VertexWeightfEXT(float weight);

void VertexWeightfvEXT(float *weight);

void VertexWeightPointerEXT(int size, enum type,
                            sizei stride, void *pointer);
```

New Tokens

Accepted by the <target> parameter of Enable:

```
VERTEX_WEIGHTING_EXT            0x8509
```

Accepted by the <mode> parameter of MatrixMode:

```
MODELVIEW0_EXT                  0x1700  (alias to MODELVIEW enumerant)
MODELVIEW1_EXT                  0x850A
```

```
    Accepted by the <pname> parameter of GetBooleanv, GetIntegerv,
    GetFloatv, and GetDoublev:

        VERTEX_WEIGHTING_EXT
        MODELVIEW0_EXT
        MODELVIEW1_EXT
        MODELVIEW0_MATRIX_EXT           0x0BA6  (alias to MODELVIEW_MATRIX)
        MODELVIEW1_MATRIX_EXT           0x8506
        CURRENT_VERTEX_WEIGHT_EXT       0x850B
        VERTEX_WEIGHT_ARRAY_EXT         0x850C
        VERTEX_WEIGHT_ARRAY_SIZE_EXT    0x850D
        VERTEX_WEIGHT_ARRAY_TYPE_EXT    0x850E
        VERTEX_WEIGHT_ARRAY_STRIDE_EXT  0x850F
        MODELVIEW0_STACK_DEPTH_EXT      0x0BA3  (alias to MODELVIEW_STACK_DEPTH)
        MODELVIEW1_STACK_DEPTH_EXT      0x8502

     Accepted by the <pname> parameter of GetPointerv:

        VERTEX_WEIGHT_ARRAY_POINTER_EXT     0x8510
```

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

-- Section 2.6. 2nd paragraph changed:

```
     "Each vertex is specified with two, three, or four coordinates.
     In addition, a current normal, current texture coordinates, current
     color, and current vertex weight may be used in processing each
     vertex."
```

-- Section 2.6. New paragraph after the 3rd paragraph:

```
     "A vertex weight is associated with each vertex.  When vertex
     weighting is enabled, this weight is used as a blending factor
     to blend the position and normals transformed by the primary and
     secondary modelview matrix transforms.  The vertex weighting
     functionality takes place completely in the "vertex / normal
     transformation" stage of Figure 2.2."
```

-- Section 2.6.3. First paragraph changed to

```
     "The only GL commands that are allowed within any Begin/End pairs are
     the commands for specifying vertex coordinates, vertex colors, normal
     coordinates, and texture coordinates (Vertex, Color, VertexWeightEXT,
     Index, Normal, TexCoord)..."
```

-- Section 2.7. New paragraph after the 4th paragraph:

```
     "The current vertex weight is set using

         void VertexWeightfEXT(float weight);
         void VertexWeightfvEXT(float *weight);

     This weight is used when vertex weighting is enabled."
```

-- Section 2.7. The last paragraph changes from

```
     "... and one floating-point value to store the current color index."
```

    to:

    "... one floating-point number to store the vertex weight, and one
    floating-point value to store the current color index."

-- Section 2.8. Change 1st paragraph to say:

    "The client may specify up to seven arrays: one each to store edge
    flags, texture coordinates, colors, color indices, vertex weights,
    normals, and vertices. The commands"

    Add to functions listed following first paragraph:

```
   void VertexWeightPointerEXT(int size, enum type,
                               sizei stride, void *pointer);
```

    Add to table 2.4 (p. 22):

Command	Sizes	Types
VertexWeightPointerEXT	1	float

    Starting with the second paragraph on p. 23, change to add
    VERTEX_WEIGHT_ARRAY_EXT:

    "An individual array is enabled or disabled by calling one of

```
       void EnableClientState(enum array)
       void DisableClientState(enum array)
```

    with array set to EDGE_FLAG_ARRAY, TEXTURE_COORD_ARRAY, COLOR_ARRAY,
    INDEX_ARRAY, VERTEX_ARRAY_WEIGHT_EXT, NORMAL_ARRAY, or VERTEX_ARRAY,
    for the edge flag, texture coordinate, color, secondary color,
    color index, normal, or vertex array, respectively.

    The ith element of every enabled array is transferred to the GL by calling

```
       void ArrayElement(int i)
```

    For each enabled array, it is as though the corresponding command
    from section 2.7 or section 2.6.2 were called with a pointer to
    element i. For the vertex array, the corresponding command is
    Vertex<size><type>v, where <size> is one of [2,3,4], and <type> is
    one of [s,i,f,d], corresponding to array types short, int, float, and
    double respectively. The corresponding commands for the edge flag,
    texture coordinate, color, secondary color, color index, and normal
    arrays are EdgeFlagv, TexCoord<size><type>v, Color<size><type>v,
    Index<type>v, VertexWeightfvEXT, and Normal<type>v, respectively..."

    Change pseudocode on p. 27 to disable vertex weight array for canned
    interleaved array formats. After the lines

```
       DisableClientState(EDGE_FLAG_ARRAY);
       DisableClientState(INDEX_ARRAY);
```

    insert the line

```
            DisableClientState(VERTEX_WEIGHT_ARRAY_EXT);
```

Substitute "seven" for every occurrence of "six" in the final paragraph on p. 27.

-- Section 2.10. Change the sentence:

"The model-view matrix is applied to these coordinates to yield eye coordinates."

to:

"The primary modelview matrix is applied to these coordinates to yield eye coordinates. When vertex weighting is enabled, a secondary modelview matrix is also applied to the vertex coordinates, the result of the two modelview transformations are weighted by its respective vertex weighting factor and combined by addition to yield the true eye coordinates. Vertex weighting is enabled or disabled using Enable and Disable (see section 2.10.3) with an argument of VERTEX_WEIGHTING_EXT."

Change the 4th paragraph to:

"If vertex weighting is disabled and a vertex in object coordinates is given by (xo yo zo wo)' and the primary model-view matrix is M0, then the vertex's eye coordinates are found as

```
   (xe ye ze we)'  =  M0 (xo yo zo wo)'
```

If vertex weighting is enabled, then the vertex's eye coordinates are found as

```
   (xe0 ye0 ze0 we0)'  =  M0 (xo yo zo wo)'

   (xe1 ye1 ze1 we1)'  =  M1 (xo yo zo wo)'

   (xe,ye,ze)' = vw*(xe0,ye0,ze0)' + (1-vw) * (xe1,ye1,ze1)'

   we = we0
```

where M1 is the secondary modelview matrix and vw is the current vertex weight."

-- Section 2.10.2 Change the 1st paragraph to say:

"The projection matrix and the primary and secondary modelview matrices are set and modified with a variety of commands. The affected matrix is determined by the current matrix mode. The current matrix mode is set with

```
   void MatrixMode(enum mode);
```

which takes one of the four pre-defined constants TEXTURE, MODELVIEW0, MODELVIEW1, or PROJECTION (note that MODELVIEW is an alias for MODELVIEW0). TEXTURE is described later. If the current matrix is MODELVIEW0, then matrix operations apply to the primary

```
    modelview matrix; if MODELVIEW1, then matrix operations apply to
    the secondary modelview matrix; if PROJECTION, then they apply to
    the projection matrix."

    Change the 9th paragraph to say:

    "There is a stack of matrices for each of the matrix modes.  For the
    MODELVIEW0 and MODELVIEW1 modes, the stack is at least 32 (that is,
    there is a stack of at least 32 modelview matrices). ..."

    Change the last paragraph to say:

    "The state required to implement transformations consists of a
    four-valued integer indicating the current matrix mode, a stack of
    at least two 4x4 matrices for each of PROJECTION and TEXTURE with
    associated stack pointers, and two stacks of at least 32 4x4 matrices
    with an associated stack pointer for MODELVIEW0 and MODELVIEW1.
    Initially, there is only one matrix on each stack, and all matrices
    are set to the identity.  The initial matrix mode is MODELVIEW0."
```

-- Section 2.10.3 Change the 2nd and 7th paragraphs to say:

```
    "For a modelview matrix M, the normal for this matrix is transformed
    to eye coordinates by:

       (nx' ny' nz' q') = (nx ny nz q) * M^-1

    where, if (x y z w)' are the associated vertex coordinates, then

           /  0,                     w= 0
           |
       q = |  -(nx ny nz) (x y z)'                         (2.1)
           |  --------------------,  w != 0
           \          w

    Implementations may choose instead to transform (x y z)' to eye
    coordinates using

       (nx' ny' nz') = (nx ny nz) * Mu^-1

    Where Mu is the upper leftmost 3x3 matrix taken from M.

    Rescale multiplies the transformed normals by a scale factor

       ( nx" ny" nz" ) = f (nx' ny' nz')

    If rescaling is disabled, then f = 1.  If rescaling is enabled, then
    f is computed as (mij denotes the matrix element in row i and column j
    of M^-1, numbering the topmost row of the matrix as row 1 and the
   leftmost column as column 1

                              1
              f =    ---------------------------
                     sqrt(m31^2 + m32^2 + m33^2)

    Note that if the normals sent to GL were unit length and the model-view
    matrix uniformly scales space, the rescale make sthe transformed normals
    unit length.

    Alternatively, an implementation may chose f as

                              1
              f =     --------------------------
                      sqrt(nx'^2 + ny'^2 + nz'^2)

    recomputing f for each normal.  This makes all non-zero length
    normals unit length regardless of their input length and the nature
    of the modelview matrix.

    After rescaling, the final transformed normal used in lighting, nf,
    depends on whether vertex weighting is enabled or not.

    When vertex weighting is disabled, nf is computed as

           nf = m * ( nx"0  ny"0  nz"0 )

    where (nx"0 ny"0 nz"0) is the normal transformed as described
    above using the primary modelview matrix for M.

    If normalization is enabled m=1.  Otherwise

                              1
              m =     -----------------------------
                      sqrt(nx"0^2 + ny"0^2 + nz"0^2)

    However when vertex weighting is enabled, the normal is transformed
    twice as described above, once by the primary modelview matrix and
    again by the secondary modelview matrix, weighted using the current
    per-vertex weight, and normalized.  So nf is computed as

           nf = m * ( nx"w  ny"w  nz"w )

    where nw is the weighting normal computed as

           nw = vw * ( nx"0  ny"0  nz"0 ) + (1-vw) * (nx"1 ny"1 nz"1)

    where (nx"0 ny"0 nz"0) is the normal transformed as described
    above using the primary modelview matrix for M, and (nx"1 ny"1 nz"1) is
    the normal transformed as described above using the secondary modelview
    matrix for M, and vw is the current pver-vertex weight."
```

-- Section 2.12. Changes the 3rd paragraph:

    "The coordinates are treated as if they were specified in a
    Vertex command. The x, y, z, and w coordinates are transformed
    by the current primary modelview and perspective matrices. These
    coordinates, along with current values, are used to generate a
    color and texture coordinates just as done for a vertex, except
    that vertex weighting is always treated as if it is disabled."

Additions to Chapter 3 of the GL Specification (Rasterization)

None

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the GL Specification (Special Functions)

None

Additions to Chapter 6 of the GL Specification (State and State Requests)

None

Additions to the GLX Specification

None

GLX Protocol

A new GL rendering command is added. The following command is sent to the server as part of a glXRender request:

```
VertexWeightfvEXT
    2  8        rendering command length
    2  4135     rendering command opcode
    4  FLOAT32  weight0
```

To support vertex arrays, the DrawArrays rendering command (sent via a glXRender or glXRenderLarge request) is amended as follows:

The list of arrays listed for the third element in the ARRAY_INFO structure is amended to include:

```
0x850c         j=1        VERTEX_WEIGHT_ARRAY_EXT
```

The VERTEX_DATA description is amended to include:

```
If the vertex weight array is enabled:
ws            LISTofBYTE        vertex weight array element
wp                              unused, wp=pad(ws)
```

with the following paragraph amended to read:

"where ns, cs, is, ts, es, vs, ws is the size of the normal, color, index, texture, edge, vertex, and vertex weight array elements and np, cp, ip, tp, ep, vp, wp is the padding for the normal, color, index, texture, edge, vertex, and vertex weight array elements, respectively."

Errors

The current vertex weight can be updated at any time. In particular WeightVertexEXT can be called between a call to Begin and the corresponding call to End.

INVALID_VALUE is generated if VertexWeightPointerEXT parameter <size> is not 1.

    INVALID_ENUM is generated if VertexWeightPointerEXT parameter <type>
    is not FLOAT.

    INVALID_VALUE is generated if VertexWeightPointerEXT parameter <stride>
    is negative.

New State

(table 6.5, p196)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
CURRENT_VERTEX_WEIGHT_EXT	F	GetFloatv	1	Current vertex weight	2.8	current

(table 6.6, p197)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
VERTEX_WEIGHT_ARRAY_EXT	B	IsEnabled	False	Vertex weight enable	2.8	vertex-array
VERTEX_WEIGHT_ARRAY_SIZE_EXT	Z+	GetIntegerv	1	Weights per vertex	2.8	vertex-array
VERTEX_WEIGHT_ARRAY_TYPE_EXT	Z1	GetIntegerv	FLOAT	Type of weights	2.8	vertex-array
VERTEX_WEIGHT_ARRAY_STRIDE_EXT	Z	GetIntegerv	0	Stride between weights	2.8	vertex-array
VERTEX_WEIGHT_ARRAY_POINTER_EXT	Y	GetPointerv	0	Pointer to vertex weight array	2.8	vertex-array

(table 6.7, p198)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
MODELVIEW0_MATRIX_EXT	32*xM4	GetFloatv	Identity	Primary modelview stack	2.10.2	-
MODELVIEW1_MATRIX_EXT	32*xM4	GetFloatv	Identity	Secondary modelview stack	2.10.2	-
MODELVIEW0_STACK_DEPTH_EXT	Z+	GetIntegerv	1	Primary modelview stack depth	2.10.2	-
MODELVIEW1_STACK_DEPTH_EXT	Z+	GetIntegerv	1	Secondary modelview stack depth	2.10.2	-
MATRIX_MODE	Z4	GetIntegerv	MODELVIEW0	Current matrix mode	2.10.2	transform
VERTEX_WEIGHTING_EXT	B	IsEnabled	False	Vertex weighting on/off	2.10.2	transform/enable

    NOTE: MODELVIEW_MATRIX is an alias for MODELVIEW0_MATRIX_EXT
           MODELVIEW_STACK_DEPTH is an alias for MODELVIEW0_STACK_DEPTH_EXT

New Implementation Dependent State

    None

Revision History

    12/16/2000 amended to include GLX protocol for vertex arrays
    5/25/2000 added missing MODELVIEW#_MATRIX tokens values

GL_HP_occlusion_test - PRELIMINARY

XXX - Not complete yet!!!

Name

    HP_occlusion_test

Name Strings

    GL_HP_occlusion_test

Number

    137

Overview

This extension defines a mechanism whereby an application can determine the non-visibility of some set of geometry based on whether an encompassing set of geometry is non-visible. In general this feature does not guarantee that the target geometry is visible when the test fails, but is accurate with regard to non-visibility.

Occlusion culling allows an application to render some geometry and at the completion of the rendering to determine if any of the geometry could or did modify the depth buffer, ie. a depth buffer test succeeded. The idea being that if the application renders a bounding box of some geometry in this mode and the occlusion test failed (ie. the bounding box was depth culled due to the current contents of the depth buffer) then the geometry enclosed by the bounding box would also be depth culled. Occlusion culling operates independently of the current rendering state (ie. when occlusion culling is enabled fragments are generated and the depth and/or color buffer may be updated). To prevent updating the depth/color buffers the application must disable updates to these buffers. As a side effect of reading the occlusion result the internal result state is cleared, setting it up for a new bounding box test.

The expected usage of this feature is :

```
    - disable updates to color and depth buffer (optional)
        glDepthMask(GL_FALSE)
        glColorMask(GL_FALSE,GL_FALSE,GL_FALSE,GL_FALSE)

    - enable occlusion test
        glEnable(GL_OCCLUSION_TEST_HP)

    - render bounding geometry
        gl rendering calls

    - disable occlusion test
        glDisable(GL_OCCLUSION_TEST_HP)

    - enable updates to color and depth buffer
        glDepthMask(GL_TRUE)
        glColorMask(GL_TRUE,GL_TRUE,GL_TRUE,GL_TRUE)
    - read occlusion test result
        glGetBooleanv(GL_OCCLUSION_TEST_RESULT_HP, &result)

    - if (result) render internal geometry
      else don't render
```

For this extension to be useful the assumption are being made :

- the time to render the geometry under test is much more than rendering the encompassing geometry, including reading back the test result

- the application is modelling data that includes occluding structures (eg. walls, hierarchial assemblies, ...)

- the application is structured in such a way as to utilize bounding boxes for encompassing geometry

New Procedures and Functions

none

New Tokens

Accepted by the <cap> parameter of Enable, Disable, and IsEnabled, by the <pname> of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev :

```
        GL_OCCLUSION_TEST_HP           0x8165
```

Accepted by the <pname> of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev :

```
        GL_OCCLUSION_TEST_RESULT_HP    0x8166
```

New State

Boolean result of occlusion test, initial value of FALSE. The result is set to FALSE as a side effect of reading it (executing a Get call).

Issue

This extension is superceded by the GL_HP_visibility_test extension.

Also see NVIDIA's NV_occlusion_query extension.

Name

    IBM_rasterpos_clip

Name Strings

    GL_IBM_rasterpos_clip

Version

    $Id: //depot/main/doc/registry/extensions/IBM/rasterpos_clip.spec#1 $

Number

    110

Dependencies

    None

Overview

    IBM_rasterpos_clip extends the semantics of the RasterPos functions. It
    provides an enable that allows a raster position that would normally be
    clipped to be treated as a valid (albeit out-of-viewport) position.

    This extension allows applications to specify geometry-aligned pixel
    primitives that may be partially off-screen. These primitives are
    tested on a pixel-by-pixel basis without being rejected completely
    because of an invalid raster position.

Issues

    Currently, clipping is disabled only in X and Y. If disabling Z
    clipping is required, the behavior needs to be specified.

New Procedures and Functions

    None

New Tokens

    Accepted by the <target> parameter of Enable and Disable and the <value>
    parameter of IsEnabled, GetBooleanv, GetIntegerv, GetFloatv, GetDoublev:

        RASTER_POSITION_UNCLIPPED_IBM   103010

    The enum is subject to change if this proposal attracts interest from
    other vendors and becomes an EXT extension.

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

    In Section 2.12, the behavior of valid bit of the raster position with
    respect to vertex clipping is defined.

        The transformed coordinates are passed to clipping as if they
        represented a point. If the "point" is not culled, then the

```
        projection to window coordinates is computed (section 2.10) and
        saved as the current raster position, and the valid bit is set.  If
        the "point" is culled, ... the valid bit is cleared.

    The specification is modified to read:

        The transformed coordinates are passed to clipping as if they
        represented a point.  If (1) the "point" is not culled, or (2)
        RASTER_POSITION_UNCLIPPED_IBM is enabled and the "point" is not culled
        except by the x and y components of the clip volume, then the
        projection to window coordinates is computed (section 2.10) and saved
        as the current raster position, and the valid bit is set.  Otherwise,
        ... the valid bit is cleared.
```

Additions to Chapter 3 of the GL Specification (Rasterization)

    None

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

    None

Additions to Chapter 5 of the GL Specification (Special Functions)

    None

Additions to Chapter 6 of the GL Specification (State and State Requests)

    None

Additions to the GLX Specification

    None

GLX Protocol

    None (other than the new Enable target).

Errors

    None

New State

```
    Get Value                     Type Get Command  Value Sec   Attrib
    ----------------------------- ---- -----------  ----- ----  -------
    RASTER_POSITION_UNCLIPPED_IBM   B  IsEnabled    False 2.12  transform/
                                                                enable
```

New Implementation Dependent State

    None

Name

    IBM_texture_mirrored_repeat

Name Strings

    GL_IBM_texture_mirrored_repeat

Version

```
    $Date: 1999/12/28 01:40:35 $ $Revision: 1.2 $
    IBM Id: texture_mirrored_repeat.spec,v 1.5 1998/01/16 18:09:31 pbrown Exp
```

Number

    224

Dependencies

```
    EXT_texture_3D
    IBM_texture_edge_clamp
```

Overview

IBM_texture_mirrored_repeat extends the set of texture wrap modes to include a mode (GL_MIRRORED_REPEAT_IBM) that effectively uses a texture map twice as large at the original image in which the additional half of the new image is a mirror image of the original image.

This new mode relaxes the need to generate images whose opposite edges match by using the original image to generate a matching "mirror image".

Issues

* The spec clamps the final (u,v) coordinates to the range [0.5, 2^n-0.5]. This will produce the same effect as trapping a sample of the border texel and using the corresponding edge texel. The choice of technique is purely an implementation detail.

New Procedures and Functions

None

New Tokens

Accepted by the <param> parameter of TexParameteri and TexParameterf, and by the <params> parameter of TexParameteriv and TexParameterfv, when their <pname> parameter is TEXTURE_WRAP_S, TEXTURE_WRAP_T, or TEXTURE_WRAP_R_EXT:

```
      GL_MIRRORED_REPEAT_IBM           0x8370
```

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

None.

Additions to Chapter 3 of the GL Specification (Rasterization)

    None

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

```
  Change to Section 3.8 (Subsection "Texture Wrap Modes")

    If TEXTURE_WRAP_S, TEXTURE_WRAP_T, or TEXTURE_WRAP_R_EXT is set to
    MIRRORED_REPEAT_IBM, the s (or t or r) coordinate is converted to:

        s - floor(s),           if floor(s) is even, or
        1 - (s - floor(s)),     if floor(s) is odd.


  Change to Section 3.8.1, Texture Minification

    Let:
        u(x,y) = 2^n * s(x,y),
        v(x,y) = 2^m * t(x,y), and
        w(x,y) = 2^l * r(x,y).

    If the TEXTURE_WRAP_S, TEXTURE_WRAP_T, or TEXTURE_WRAP_R_EXT is set to
    either MIRRORED_REPEAT_IBM or CLAMP_TO_EDGE_IBM, the resulting u, v, or
    w coordinates (respectively) are clamped to the range [0.5, 2^n-0.5].
```

Additions to Chapter 5 of the GL Specification (Special Functions)

    None

Additions to Chapter 6 of the GL Specification (State and State Requests)

    None

Additions to the GLX Specification

    None

GLX Protocol

    None.

Errors

    None

Dependencies on EXT_texture3D

    If EXT_texture3D is not implemented, then the references clamping of 3D
    textures in this file are invalid, and references to TEXTURE_WRAP_R_EXT
    should be ignored.

Dependencies on IBM_texture_edge_clamp

    If IBM_texture_edge_clamp is not implemented, then the references to
    CLAMP_TO_EDGE_IBM should be ignored.

New State

Only the type information changes for these parameters:

Get Value	Get Command	Type	Initial Value	Attrib
TEXTURE_WRAP_S	GetTexParameteriv	n x Z5	REPEAT	texture
TEXTURE_WRAP_T	GetTexParameteriv	n x Z5	REPEAT	texture
TEXTURE_WRAP_R_EXT	GetTexParameteriv	n x Z5	REPEAT	texture

New Implementation Dependent State

None

Name

    NV_blend_square

Name Strings

    GL_NV_blend_square

Version

    Date: 8/7/1999 Version: 1.0

Number

    194

Dependencies

    Written based on the wording of the OpenGL 1.2 specification.

Overview

    It is useful to be able to multiply a number by itself in the blending
    stages -- for example, in certain types of specular lighting effects
    where a result from a dot product needs to be taken to a high power.

    This extension provides four additional blending factors to permit
    this and other effects: SRC_COLOR and ONE_MINUS_SRC_COLOR for source
    blending factors, and DST_COLOR and ONE_MINUS_DST_COLOR for destination
    blending factors.

New Procedures and Functions

    None

New Tokens

    None

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

    None

Additions to Chapter 3 of the GL Specification (Rasterization)

    None

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

Two lines are added to each of tables 4.1 and 4.2:

Value	Blend Factors	
ZERO	(0, 0, 0, 0)	
ONE	(1, 1, 1, 1)	
SRC_COLOR	(Rs, Gs, Bs, As)	NEW
ONE_MINUS_SRC_COLOR	(1, 1, 1, 1) - (Rs, Gs, Bs, As)	NEW
DST_COLOR	(Rd, Gd, Bd, Ad)	
ONE_MINUS_DST_COLOR	(1, 1, 1, 1) - (Rd, Gd, Bd, Ad)	
SRC_ALPHA	(As, As, As, As) / Ka	
ONE_MINUS_SRC_ALPHA	(1, 1, 1, 1) - (As, As, As, As) / Ka	
DST_ALPHA	(Ad, Ad, Ad, Ad) / Ka	
ONE_MINUS_DST_ALPHA	(1, 1, 1, 1) - (Ad, Ad, Ad, Ad) / Ka	
CONSTANT_COLOR	(Rc, Gc, Bc, Ac)	
ONE_MINUS_CONSTANT_COLOR	(1, 1, 1, 1) - (Rc, Gc, Bc, Ac)	
CONSTANT_ALPHA	(Ac, Ac, Ac, Ac)	
ONE_MINUS_CONSTANT_ALPHA	(1, 1, 1, 1) - (Ac, Ac, Ac, Ac)	
SRC_ALPHA_SATURATE	(f, f, f, 1)	

Table 4.1: Values controlling the source blending function and the source blending values they compute. f = min(As, 1 - Ad).

Value	Blend Factors	
ZERO	(0, 0, 0, 0)	
ONE	(1, 1, 1, 1)	
SRC_COLOR	(Rs, Gs, Bs, As)	
ONE_MINUS_SRC_COLOR	(1, 1, 1, 1) - (Rs, Gs, Bs, As)	
DST_COLOR	(Rd, Gd, Bd, Ad)	NEW
ONE_MINUS_DST_COLOR	(1, 1, 1, 1) - (Rd, Gd, Bd, Ad)	NEW
SRC_ALPHA	(As, As, As, As) / Ka	
ONE_MINUS_SRC_ALPHA	(1, 1, 1, 1) - (As, As, As, As) / Ka	
DST_ALPHA	(Ad, Ad, Ad, Ad) / Ka	
ONE_MINUS_DST_ALPHA	(1, 1, 1, 1) - (Ad, Ad, Ad, Ad) / Ka	
CONSTANT_COLOR_EXT	(Rc, Gc, Bc, Ac)	
ONE_MINUS_CONSTANT_COLOR_EXT	(1, 1, 1, 1) - (Rc, Gc, Bc, Ac)	
CONSTANT_ALPHA_EXT	(Ac, Ac, Ac, Ac)	
ONE_MINUS_CONSTANT_ALPHA_EXT	(1, 1, 1, 1) - (Ac, Ac, Ac, Ac)	

Table 4.2: Values controlling the destination blending function and the destination blending values they compute.

Additions to Chapter 5 of the GL Specification (Special Functions)

None

Additions to Chapter 6 of the GL Specification (State and State Requests)

None

Additions to the GLX Specification

None

GLX Protocol

None

Errors

None

New State

(table 6.15, page 205)

Get Value	Type	Get Command	Initial Value	Sec	Attribute
BLEND_SRC	Z15	GetIntegerv	ONE	4.1.6	color-buffer
BLEND_DST	Z14	GetIntegerv	ZERO	4.1.6	color-buffer

NOTE: the only change is that Z13 changes to Z15 and Z12 changes to Z14

New Implementation Dependent State

None

Name

NV_conditional_render

Name Strings

GL_NV_conditional_render

Status

Shipping.

Version

```
Last Modified Date:         11/29/2007
NVIDIA Revision:            2
```

Number

Unassigned.

Dependencies

The extension is written against the OpenGL 2.0 Specification.

ARB_occlusion_query or OpenGL 1.5 is required.

Overview

This extension provides support for conditional rendering based on the results of an occlusion query. This mechanism allows an application to potentially reduce the latency between the completion of an occlusion query and the rendering commands depending on its result. It additionally allows the decision of whether to render to be made without application intervention.

This extension defines two new functions, BeginConditionalRenderNV and EndConditionalRenderNV, between which rendering commands may be discarded based on the results of an occlusion query. If the specified occlusion query returns a non-zero value, rendering commands between these calls are executed. If the occlusion query returns a value of zero, all rendering commands between the calls are discarded.

If the occlusion query results are not available when BeginConditionalRenderNV is executed, the <mode> parameter specifies whether the GL should wait for the query to complete or should simply render the subsequent geometry unconditionally.

Additionally, the extension provides a set of "by region" modes, allowing for implementations that divide rendering work by screen regions to perform the conditional query test on a region-by-region basis without checking the query results from other regions. Such a mode is useful for cases like split-frame SLI, where a frame is divided between multiple GPUs, each of which has its own occlusion query hardware.

New Procedures and Functions

```
    void BeginConditionalRenderNV(uint id, enum mode);
    void EndConditionalRenderNV(void);
```

New Tokens

    Accepted by the <mode> parameter of BeginConditionalRenderNV:

```
        QUERY_WAIT_NV                                   0x8E13
        QUERY_NO_WAIT_NV                                0x8E14
        QUERY_BY_REGION_WAIT_NV                         0x8E15
        QUERY_BY_REGION_NO_WAIT_NV                      0x8E16
```

Additions to Chapter 2 of the OpenGL 2.0 Specification (OpenGL Operation)

(Incorporate the spec edits from the EXT_transform_feedback specification that move the "Occlusion Queries" Section 4.1.7 -- to between Section 2.11, Coordinate Transforms and Section 2.12, Clipping, and rename it to "Asynchronous Queries". Insert a new section immediately after the moved "Asynchronous Queries" section. If EXT_transform_feedback is incorporated, this section should be inserted prior the the "Transform Feedback" section.)

Section 2.X, Conditional Rendering

Conditional rendering can be used to discard rendering commands based on the result of an occlusion query. Conditional rendering is started and stopped using the commands

```
    void BeginConditionalRenderNV(uint id, enum mode);
    void EndConditionalRenderNV(void);
```

<id> specifies the name of an occlusion query object whose results are used to determine if the rendering commands are discarded. If the result (SAMPLES_PASSED) of the query is zero, all rendering commands between BeginConditionalRenderNV and the corresponding EndConditionalRenderNV are discarded. In this case, Begin, End, all vertex array commands performing an implicit Begin and End, DrawPixels (section 3.6), Bitmap (section 3.7), Clear (section 4.2.3), Accum (section 4.2.4), CopyPixels (section 4.3.3), EvalMesh1, and EvalMesh2 (section 5.1) have no effect. The effect of commands setting current vertex state (e.g., Color, VertexAttrib) is undefined. If the result of the occlusion query is non-zero, such commands are not discarded.

<mode> specifies how BeginConditionalRenderNV interprets the results of the occlusion query given by <id>. If <mode> is QUERY_WAIT_NV, the GL waits for the results of the query to be available and then uses the results to determine if subsquent rendering commands are discarded. If <mode> is QUERY_NO_WAIT_NV, the GL may choose to unconditionally execute the subsequent rendering commands without waiting for the query to complete.

If <mode> is QUERY_BY_REGION_WAIT_NV, the GL will also wait for occlusion query results and discard rendering commands if the result of the occlusion query is zero. If the query result is non-zero, subsequent rendering commands are executed, but the GL may discard the results of the commands for any region of the framebuffer that did not contribute to the sample count in the specified occlusion query. Any such discarding is done in an implementation-dependent manner, but the rendering command results may not be discarded for any samples that contributed to the occlusion query sample count. If <mode> is QUERY_BY_REGION_NO_WAIT_NV, the GL operates as in QUERY_BY_REGION_WAIT_NV, but may choose to unconditionally execute the subsequent rendering commands without waiting for the query to complete.

If BeginConditionalRenderNV is called while conditional rendering is in progress, or if EndConditionalRenderNV is called while conditional rendering is not in progress, the error INVALID_OPERATION is generated. The error INVALID_VALUE is generated if <id> is not the name of an existing query object query. The error INVALID_OPERATION is generated if <id> is the name of a query object with a target other than SAMPLES_PASSED, or <id> is the name of a query currently in progress.

Additions to Chapter 3 of the OpenGL 2.0 Specification (Rasterization)

None.

Additions to Chapter 4 of the OpenGL 2.0 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 2.0 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 2.0 Specification (State and State Requests)

None.

Additions to Appendix A of the OpenGL 2.0 Specification (Invariance)

None.

Additions to the AGL/GLX/WGL Specifications

None.

GLX Protocol

TBD.

Errors

INVALID_OPERATION is generated by BeginConditionalRenderNV if a previous BeginConditionalRenderNV command has been executed without a corresponding EndConditionalRenderNV command.

INVALID_OPERATION is generated by EndConditionalRenderNV if no corresponding BeginConditionalRenderNV command has been executed.

    INVALID_VALUE is generated by BeginConditionalRenderNV if <id> is not the
    name of an existing occlusion query object.

    INVALID_OPERATION is generated by BeginConditionalRenderNV if <id> is the
    name of a query object with a <target> other than SAMPLES_PASSED.

    INVALID_OPERAITON is generated by BeginConditionalRenderNV if the query
    identified by <id> is still in progress.

Issues

(1) How should rendering commands other than "normal" Begin/End-style geometry be affected by conditional rendering?

RESOLVED: All rendering commands (DrawPixels, Bitmap, Clear, Accum, etc...) are performed conditionally.

(2) What does NO_WAIT do, and why would anyone care?

RESOLVED: Hardware OpenGL implementations are heavily pipelined. After vertices are transformed, they are assembled into primitives and rasterized. While a GPU is rasterizing a primitive, it may be simultaneously transforming the vertices of the next primitive provided to the GL. At the same time, the CPU may be preparing hardware commands to process primitives following that one.

Conditional rendering uses the results of rasterizing one primitive (an occlusion query) to determine whether it will process subsequent ones. In a pipelined implementation, the initial set of primitives may not be finished drawing by the time the GL needs the occlusion query results. Waiting for the query results will leave portions of the GPU temporarily idle. It may be preferable to avoid the idle time by proceeding with a conservative assumption that the primitives rendered during the occlusion query will hit at least one sample. The NO_WAIT <mode> parameter tells the driver move ahead in that case.

For best performance, applications should attempt to insert some amount of non-dependent rendering between an occlusion query and the conditionally-rendered primitives that depend on the query result.

(3) What does BY_REGION do, and why should anyone care?

RESOLVED: Conditional rendering may be used for a variety of effects. Some of these use conditional rendering only for performance. One common use would be to draw a bounding box for a primitive unconditionally with an occlusion query active, and then conditionally execute a DrawElements call to draw the full (complex) primitive. If the bounding box is not visible, any work needed to process the full primitive can be skipped in the conditional rendering pass.

In a split-screen SLI implementation, one GPU might draw the top half of the scene while a second might draw the bottom half. The results of the occlusion query would normally be obtained by combining individual occlusion query results from each half of the screen. However, it is not necessary to do this for the bounding box algorithm. We could skip this synchronization point, and each region could instead use only its local occlusion query results. If the bounding box hits only the bottom half of the screen, the complex primitive need not be drawn on the top half, because that portion is known not to be visible. The bottom half would still be drawn, but the GPU used for the top half could skip it and start drawing the next primitive specified. The QUERY_BY_REGION_*_NV modes would be useful in that case.

However, some algorithms may require conditional rendering for correctness. For example, an application may want to render a post-processing effect that should be drawn if and only if a point is visible in the scene. Drawing only half of such an effect due to BY_REGION tests would not be desirable.

For QUERY_BY_REGION_NO_WAIT_NV, we expect that GL implementations using region-based rendering will discard rendering commands in any region where query results are available and the region's sample count is zero. Rendering would proceed normally in all other regions. The spec language doesn't require such behavior, however.

(4) Should the <mode> parameter passed to BeginConditionalRenderNV be specified as a hint instead?

RESOLVED: The "wait" or "don't wait" portion of the <mode> parameter could be a hint. But it doesn't fit nicely with the FASTEST or NICEST values that are normally passed to Hint. Providing this functionality via a <mode> parameter to BeginConditionalRenderNV seems to make the most sense. Note that the <mode> parameter is specified such that QUERY_NO_WAIT_NV can be implemented as though QUERY_WAIT_NV were specified, which makes the "NO_WAIT" part of the mode a hint.

The "BY_REGION" part is also effectively a hint. These modes may be implemented as though the equivalent non-BY_REGION mode were provided. Many OpenGL implementations will do all of their processing in a single region.

(5) What happens if BeginQuery is called while the specified occlusion query is begin used for conditional rendering?

RESOLVED: An INVALID_OPERATION error is generated.

(6) Should conditional rendering work with any type of query other than SAMPLES_PASSED (occlusion)?

RESOLVED: Not in this extension. The spec currently requires that <id> be the name of an occlusion query. There might be other query types where such an operation would make sense, but there aren't any in the current OpenGL spec.

(7) What is the effect on current state for immediate mode attribute calls (e.g., Color, VertexAttrib) made during conditional rendering if the corresponding occlusion query failed?

RESOLVED: The effect of these calls is undefined. If subsequent primitives depend on a vertex attribute set inside a conditional rendering block, and application should re-send the values after EndConditionalRenderNV.

(8) Should we provide any new query object types for conditional rendering?

RESOLVED: No. It may be useful to some GL implementations to provide an occlusion query type that only returns "zero" or "non-zero", or to provide a query type that is used only for conditional rendering but doesn't have to maintain results that can be returned to the application. However, performing conditional rendering using only the occlusion query mechanisms already in core OpenGL is sufficient for the platforms targeted by this extension.

(9) What happens if QUERY_BY_REGION_* is used, and the application switches between windows or FBOs between the occlusion query and conditional rendering blocks? The "regions" used for the two operations may not be identical.

RESOLVED: The spec language doesn't specifically address this issue, and implementations may choose to define regions arbitrarily in this case.

We strongly recommend that applications using QUERY_BY_REGION_* should not change windows or FBO configuration between the occlusion query and the dependent rendering.

Usage Example

```
GLuint queryID = 0x12345678;

// Use an occlusion query while rendering the bounding box of the real
// object.
glBeginQuery(GL_SAMPLES_PASSED, queryID);
   drawBoundingBox();
glEndQuery(GL_SAMPLES_PASSED);

// Do some unrelated rendering in hope that the query result will be
// available by the time we call glBeginConditionalRenderNV.

// Now conditionally render the real object if any portion of its
// bounding box is visible.
glBeginConditionalRenderNV(queryID, GL_QUERY_WAIT_NV);
   drawComplicatedObject();
glEndConditionalRenderNV();
```

Revision History

```
Rev.    Date    Author    Changes
----  --------  --------  -----------------------------------------
 2    11/29/07  ewerness  First public release
 1                        Internal revisions
```

Name

    NV_copy_depth_to_color

Name Strings

    GL_NV_copy_depth_to_color

Notice

IP Status

NVIDIA Proprietary.

Status

Shipping (version 1.0)

Version

NVIDIA Date: October 17, 2001 (version 1.0)

Number

243

Dependencies

Written based on the wording of the OpenGL 1.2.1 specification.

Requires support for the NV_packed_depth_stencil extension.

Overview

Some applications, especially systems for distributed OpenGL rendering, would like to have a fast way of copying their depth buffer into a color buffer; for example, this allows the depth buffer to be scanned out, allowing downstream compositing operations.

To do this operation in unextended OpenGL, the app must use glReadPixels of GL_DEPTH_COMPONENT data, followed by glDrawPixels of RGBA data. However, this typically will not provide adequate performance.

This extension provides a way to copy the depth data directly into the color buffer, by adding two new options for the "type" parameter of glCopyPixels: GL_DEPTH_STENCIL_TO_RGBA_NV and GL_DEPTH_STENCIL_TO_BGRA_NV.

Typically, OpenGL implementations support many more bits of depth precision than color precision per channel. On many PC platforms, it is common, for example, to have 24 bits of depth, 8 bits of stencil, and 8 bits of red, green, blue, and alpha.

In such a framebuffer configuration, the most effective way to copy the data without this extension would be to perform a glReadPixels of GL_UNSIGNED_INT_24_8_NV/GL_DEPTH_STENCIL_NV (using the existing NV_packed_depth_stencil extension), followed by a glDrawPixels of GL_UNSIGNED_INT_8_8_8_8/GL_RGBA or GL_BGRA data. This places the depth data in the color channels and the stencil data in the alpha channel.

This extension's new operations concatenates these two operations, providing a CopyPixels command that does both of these steps in one. This provides a large performance speedup, since no pixel data must be transfered across the bus.

Issues

*   *Does this spec need a dependency on NV_packed_depth_stencil?*

    RESOLVED: It doesn't need it, but it does. It makes the spec a whole lot easier to write. In theory, this extension can be supported without support for NV_packed_depth_stencil; in practice, it is very unlikely that any implementation will ever support this extension, but not NV_packed_depth_stencil.

*   *Should we support copies to both RGBA and BGRA?*

    RESOLVED: Yes. We support this, so there is no reason not to allow users to choose.

*   *Should pixel transfer operations, fragment operations, and PixelZoom be applied on the new CopyPixels operations?*

    RESOLVED: Yes. This is really just a different source data type for a CopyPixels of COLOR data, so, even though the typical usage case of this extension differs, there is little reason to cripple the spec with a nonorthogonality here.

*   *What is the interaction with depth testing and stencil testing?*

    RESOLVED: They are allowed. This means that there are read-modify-write hazards with overlapping CopyPixels, but they are no worse than with other forms of overlapping CopyPixels; the rule remains that (effectively) all source data must be read before any fragments are generated.

    That having been said, it is anticipated that applications would turn these off before performing the copy, because they would likely impact performance on many implementations, especially if the source and destination regions overlapped.

*   *Should a mode useful for 16-bit depth buffers be supported?*

    RESOLVED: No, that seems fairly uninteresting.

* *What restrictions should apply to the use of this extension, both in terms of the current color buffer format and the current depth buffer format?*

RESOLVED: None beyond the requirement that the drawable must have both a depth buffer and a stencil buffer. This is similar to the behavior chosen in NV_packed_depth_stencil. For example, a ReadPixels of DEPTH_STENCIL_NV data is supported, even if the drawable does not have 24 bits of depth and 8 bits of stencil. Although it is not anticipated that this extension will be useful in other modes, it is specified to work nonetheless.

* *What useful things can be done with the stencil in the alpha?*

Although it is mostly meaningless to try to blend using the stencil, one useful way of using this feature is to use the alpha test. This allows the app to kill certain pixels based on the stencil during this operation. The app could clear the color buffer first, creating a "background" depth value, and then the CopyPixels pass could overwrite that on selected pixels.

New Procedures and Functions

None.

New Tokens

Accepted by the <type> parameter of CopyPixels:

```
DEPTH_STENCIL_TO_RGBA_NV                      0x886E
DEPTH_STENCIL_TO_BGRA_NV                      0x886F
```

Additions to Chapter 2 of the OpenGL 1.2.1 Specification (OpenGL Operation)

None.

Additions to Chapter 3 of the OpenGL 1.2.1 Specification (Rasterization)

None.

Additions to Chapter 4 of the OpenGL 1.2.1 Specification (Per-Fragment Operations and the Frame Buffer)

Update the second, third, and fourth paragraphs of section 4.3.3 (page 162) to say:

"<type> is a symbolic constant that must be one of COLOR, STENCIL, DEPTH, or DEPTH_STENCIL_NV, indicating that the values to be transfered are colors, stencil values, depth values, or depth/stencil pairs, respectively. The first four arguments have the same interpretation as the corresponding arguments to ReadPixels.

Values are obtained from the framebuffer, converted (if appropriate), then subjected to the pixel transfer operations described in section 3.6.5, just as if ReadPixels were called with the corresponding arguments. If the <type> is STENCIL or DEPTH, then it is as if the <format> for ReadPixels were STENCIL_INDEX or DEPTH_COMPONENT, respectively. If the <type> is COLOR, then if the GL is in RGBA mode, it is as if the <format> were RGBA, while if the GL is in color index mode, it is as if the <format> were COLOR_INDEX. If the <type> is any of DEPTH_STENCIL_NV, DEPTH_STENCIL_TO_RGBA_NV, or DEPTH_STENCIL_TO_BGRA_NV, it is as if the <format> were DEPTH_STENCIL_NV.

The groups of elements so obtained are then written to the framebuffer just as if DrawPixels had been given <width> and <height>, beginning with final conversion of elements. The effective <format> is the same as that already described, unless <type> is DEPTH_STENCIL_TO_RGBA_NV or DEPTH_STENCIL_TO_BGRA_NV. In that case, first, the groups of elements are packed into pixel groups of type UNSIGNED_INT_24_8_NV. Then, if <type> is DEPTH_STENCIL_TO_RGBA_NV, they are unpacked as if their type was UNSIGNED_INT_8_8_8_8 and their format was RGBA, and if <type> is DEPTH_STENCIL_TO_BGRA_NV, they are unpacked as if their type was UNSIGNED_INT_8_8_8_8 and their format was BGRA. In either case, the effective <format> of the pixels to be written to the framebuffer is RGBA."

Additions to Chapter 5 of the OpenGL 1.2.1 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 1.2.1 Specification (State and State Requests)

None.

GLX Protocol

None.

Errors

The error INVALID_OPERATION is generated if CopyPixels is called where type is DEPTH_STENCIL_TO_RGBA_NV or DEPTH_STENCIL_TO_BGRA_NV and there is not both a depth buffer and a stencil buffer.

The error INVALID_OPERATION is generated if CopyPixels is called where type is DEPTH_STENCIL_TO_RGBA_NV or DEPTH_STENCIL_TO_BGRA_NV and the GL is in color index mode.

New State

None.

Revision History

none yet

Name

    NV_depth_buffer_float

Name Strings

    GL_NV_depth_buffer_float

Contributors

    Pat Brown
    Mike Strauss

Contact

    Mike Strauss, NVIDIA Corporation (mstrauss 'at' nvidia.com)

Status

    Shipping for GeForce 8 Series (November 2006)

Version

```
    Last Modified Date:         11/06/2006
    NVIDIA Revision:            8
```

Number

    334

Dependencies

    OpenGL 2.0 is required.

    ARB_color_buffer_float is required.

    EXT_packed_depth_stencil is required.

    EXT_framebuffer_object is required.

    This extension modifies EXT_depth_bounds_test.

    This extension modifies NV_copy_depth_to_color.

    This extension is written against the OpenGL 2.0 specification.

Overview

    This extension provides new texture internal formats whose depth
    components are stored as 32-bit floating-point values, rather than the
    normalized unsigned integers used in existing depth formats.
    Floating-point depth textures support all the functionality supported for
    fixed-point depth textures, including shadow mapping and rendering support
    via EXT_framebuffer_object. Floating-point depth textures can store
    values outside the range [0,1].

By default, OpenGL entry points taking depth values implicitly clamp the values to the range [0,1]. This extension provides new DepthClear, DepthRange, and DepthBoundsEXT entry points that allow applications to specify depth values that are not clamped.

Additionally, this extension provides new packed depth/stencil pixel formats (see EXT_packed_depth_stencil) that have 64-bit pixels consisting of a 32-bit floating-point depth value, 8 bits of stencil, and 24 unused bites. A packed depth/stencil texture internal format is also provided.

This extension does not provide support for WGL or GLX pixel formats with floating-point depth buffers. The existing (but not commonly used) WGL_EXT_depth_float extension could be used for this purpose.

New Procedures and Functions

```
void DepthRangedNV(double n, double f);
void ClearDepthdNV(double d);
void DepthBoundsdNV(double zmin, double zmax);
```

New Tokens

Accepted by the <internalformat> parameter of TexImage1D, TexImage2D, TexImage3D, CopyTexImage1D, CopyTexImage2D, and RenderbufferStorageEXT, and returned in the <data> parameter of GetTexLevelParameter and GetRenderbufferParameterivEXT:

```
    DEPTH_COMPONENT32F_NV                          0x8DAB
    DEPTH32F_STENCIL8_NV                           0x8DAC
```

Accepted by the <type> parameter of DrawPixels, ReadPixels, TexImage1D, TexImage2D, TexImage3D, TexSubImage1D, TexSubImage2D, TexSubImage3D, and GetTexImage:

```
    FLOAT_32_UNSIGNED_INT_24_8_REV_NV              0x8DAD
```

Accepted by the <pname> parameters of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

```
    DEPTH_BUFFER_FLOAT_MODE_NV                     0x8DAF
```

Additions to Chapter 2 of the OpenGL 2.0 Specification (OpenGL Operation)

Modify Section 2.11.1 (Controling the Viewport), p. 41

(modify second paragraph) The factor and offset applied to z_d encoded by n and f are set using

```
    void DepthRange(clampd n, clampd f);
    void DepthRangedNV(double n, double f);
```

z_w is represented as either fixed-point or floating-point depending on whether the framebuffer's depth buffer uses fixed-point or floating-point representation. If the depth buffer uses fixed-point representation, we assume that the representation used represents each value k/(2^m - 1), where k is in {0,1,...,2^m-1}, as k (e.g. 1.0 is represented in binary as a string of all ones). The parameters n and f are clamped to [0, 1] when using DepthRange, but not when using DepthRangedNV. When n and f are applied to z_d, they are clamped to the range appropriate given the depth buffer's representation.

Additions to Chapter 3 of the OpenGL 2.0 Specification (Rasterization)

Modify Section 3.5.5 (Depth Offset), p. 112

(modify third paragraph) The minimum resolvable difference r is an implementation dependent parameter that depends on the depth buffer representation. It is the smallest difference in window coordinate z values that is guaranteed to remain distinct throughout polygon rasterization and in the depth buffer. All pairs of fragments generated by the rasterization of two polygons with otherwise identical vertices, but z_w values that differ by r, will have distinct depth values.

For fixed-point depth buffer representations, r is constant throughout the range of the entire depth buffer. For floating-point depth buffers, there is no single minimum resolvable difference. In this case, the minimum resolvable difference for a given polygon is dependent on the maximum exponent, e, in the range of z values spanned by the primitive. If n is the number of bits in the floating-point mantissa, the minimum resolvable difference, r, for the given primitive is defined as

```
    r = 2^(e - n).                                        (3.11)
```

(modify fourth paragraph) The offset value o for a polygon is

```
    o = m * factor + r * units.                           (3.12)
```

m is computed as described above. If the depth buffer uses a fixed-point representation, m is a function of depth values in the range [0, 1], and o is applied to depth values in the same range.

(modify last paragraph) For fixed-point depth buffers, fragment depth values are always limited to the range [0, 1], either by clamping after offset addition is performed (preferred), or by clamping the vertex values used in the rasterization of the polygons. Fragment depth values are not clamped when the depth buffer uses a floating-point representation.

Add a row to table 3.5, p. 128

type Parameter	GL Type	Special
...	...	...
FLOAT_32_UNSIGNED_INT_24_8_REV_NV	N/A	Yes
...	...	...

Modify Section 3.6.4 (Rasterization of Pixel Rectangles), p. 128

(modify second paragraph as updated by EXT_packed_depth_stencil) ... If the GL is in color index mode and <format> is not one of COLOR_INDEX, STENCIL_INDEX, DEPTH_COMPONENT, or DEPTH_STENCIL_EXT, then the error INVALID_OPERATION occurs. If <type> is BITMAP and <format> is not COLOR_INDEX or STENCIL_INDEX then the error INVALID_ENUM occurs. If <format> is DEPTH_STENCIL_EXT and <type> is not UNSIGNED_INT_24_8_EXT or FLOAT_32_UNSIGNED_INT_24_8_REV_NV, then the error INVALID_ENUM occurs. Some additional constraints on the combinations of <format> and <type> values that are accepted are discussed below.

(modify fifth paragraph of "Unpacking," p 130. as updated by EXT_packed_depth_stencil) Calling DrawPixels with a <type> of UNSIGNED_BYTE_3_3_2, ..., UNSIGNED_INT_2_10_10_10_REV, or UNSIGNED_INT_24_8_EXT is a special case in which all the components of each group are packed into a single unsigned byte, unsigned short, or unsigned int, depending on the type. If <type> is FLOAT_32_UNSIGNED_INT_24_8_REV_NV, the components of each group are two 32-bit words. The first word contains the float component. The second word contains packed 24-bit and 8-bit components.

Add two rows to table 3.8, p. 132

type Parameter	GL Type	Components	Pixel Formats
...	...	...	...
FLOAT_32_UNSIGNED_INT_24_8_REV_NV	N/A	2	DEPTH_STENCIL_EXT
...	...	...	...

Add a row to table 3.11, p. 134

FLOAT_32_UNSIGNED_INT_24_8_REV_NV:

```
   31 30 29 28 ... 4 3 2 1 0    31 30 29 ... 9 8 7 6 5 ... 2 1 0
  +-------------------------+  +-------------------------------+
  |    Float Component      |  | 2nd Component  | 1st Component |
  +-------------------------+  +-------------------------------+
```

(modify last paragraph of "Final Conversion," p. 136) For a depth component, an element is processed according to the depth buffer's representation. For fixed-point depth buffers, the element is first clamped to [0, 1] and then converted to fixed-point as if it were a window z value (see section 2.11.1, Controling the Viewport). Clamping and conversion are not necessary when the depth buffer uses a floating-point representation.

Modify Section 3.8.1 (Texture Image Specification), p. 150

(modify the second paragraph, p. 151, as modified by ARB_color_buffer_float) The selected groups are processed exactly as for DrawPixels, stopping just before final conversion. Each R, G, B, A, or depth value so generated is clamped based on the component type in the <internalFormat>. Fixed-point components are clamped to [0, 1]. Floating-point components are clamped to the limits of the range representable by their format. 32-bit floating-point components are in the standard IEEE float format. 16-bit floating-point components have 1 sign bit, 5 exponent bits, and 10 mantissa bits. Stencil index values are masked by 2^n-1 where n is the number of stencil bits in the internal format resolution (see below). If the base internal format is

DEPTH_STENCIL_EXT and <format> is not DEPTH_STENCIL_EXT, then the values of the stencil index texture components are undefined.

Add two rows to table 3.16, p. 154

Sized Internal Format	Base InternalFormat	R bits	G bits	B bits	A bits	L bits	I bits	D bits	S bits
...	...	...	...	...	...	...	...	...	...
DEPTH_COMPONENT32F_NV	DEPTH_COMPONENT							f32	
DEPTH32F_STENCIL8_NV	DEPTH_STENCIL_EXT							f32	8
...	...	...	...	...	...	...	...	...	...

Modify Section 3.8.14 (Texture Comparison Modes), p. 185

(modify second paragraph of "Depth Texture Comparison Mode," p. 188) Let D_t be the depth texture value, and R be the interpolated texture coordinate. If the texture's internal format indicates a fixed-point depth texture, then D_t and R are clamped to [0, 1], otherwise no clamping is performed. The effective texture value L_t, I_t, or A_t is computed as follows:

Modify Section 3.11.2 (Shader Execution), p. 194

(modify first paragraph of "Shader Outputs," p, 196, as modified by ARB_color_buffer_float) The OpenGL Shading Language specification describes the values that may be output by a fragment shader. These are gl_FragColor, gl_FragData[n], and gl_FragDepth. If fragment clamping is enabled, the final fragment color values or the final fragment data values written by a fragment shader are clamped to the range [0, 1] and then may be converted to fixed-point as described in section 2.14.9. If fragment clamping is disabled, the final fragment color values or the final fragment data values are not modified. For fixed-point depth buffers the final fragment depth written by a fragment shader is first clamped to [0, 1] and then converted to fixed-point as if it were a window z value (see section 2.11.1). Clamping and conversion are not applied for floating-point depth buffers. Note that the depth range computation is not applied here.

Additions to Chapter 4 of the OpenGL 2.0 Specification (Per-Fragment Operations and the Frame Buffer)

(modify third paragraph in the introduction, p. 198, as modified by ARB_color_buffer_float) Color buffers consist of either unsigned integer color indices, R, G, B and optionally A unsigned integer values, or R, G, B, and optionally A floating-point values. Depth buffers consist of either unsigned integer values of the format described in section 2.11.1, or floating-point values. The number of bitplanes...

Modify Section 4.2.3 (Clearing the Buffers), p. 215

(modify fourth paragraph)

The functions

```
    void ClearDepth(clampd d);
    void ClearDepthdNV(double d);
```

are used to set the depth value used when clearing the depth buffer. ClearDepth takes a floating-point value that is clamped to the range [0, 1]. ClearDepthdNV takes a floating-point value that is not clamped. When clearing a fixed-point depth buffer, the depth clear value is clamped to the range [0, 1], and converted to fixed-point according to the rules for a window z value given in section 2.11.1. No clamping or conversion are applied when clearing a floating-point depth buffer.

Modify Section 4.3.1 (Writing to the Stencil Buffer), p. 218

(modify paragraph added by EXT_packed_depth_stencil, p. 219) If the <format> is DEPTH_STENCIL_EXT, then values are taken from both the depth buffer and the stencil buffer. If there is no depth buffer or if there is no stencil buffer, then the error INVALID_OPERATION occurs. If the <type> parameter is not UNSIGNED_INT_24_8_EXT, or FLOAT_32_UNSIGNED_INT_24_8_NV then the error INVALID_ENUM occurs.

Modify Section 4.3.2 (Reading Pixels), p. 219

(modify "Conversion of Depth values," p. 222, as modified by EXT_packed_depth_stencil) This step only applies if <format> is DEPTH_COMPONENT or DEPTH_STENCIL_EXT and the depth buffer uses a fixed-point representation. An element taken from the depth buffer is taken to be a fixed-point value in [0, 1] with m bits, where m is the number of bits in the depth buffer (see section 2.11.1). No conversion is necessary if <format> is DEPTH_COMPONENT or DEPTH_STENCIL_EXT and the depth buffer uses a floating-point representation.

Add a row to table 4.6, p. 223

type Parameter	Index Mask
...	...
FLOAT_32_UNSIGNED_INT_24_8_REV_NV	2^8-1

Add a row to table 4.7, p. 224

type Parameter	GL Type	Component Conversion
...	...	...
FLOAT_32_UNSIGNED_INT_24_8_REV_NV	float	c = f (depth only)

Additions to Chapter 5 of the OpenGL 2.0 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 2.0 Specification (State and State Requests)

Modify DEPTH_RANGE entry in table 6.9 (Transformation State) p. 270

Get Value	Type	Get Command	Init Value	Description	Sec.	Attribute
DEPTH_RANGE	2xR	GetFloatv	0,1	Depth range near & far	2.11.1	viewport

Modify DEPTH_BOUNDS_EXT entry in table 6.19 (Pixel Operation) p. 280

Get Value	Type	Get Command	Init Value	Description	Sec	Attribute
DEPTH_BOUNDS_EXT	2xR	GetFloatv	0,1	Depth bounds zmin & zmax	4.1.X	depth-buffer

Modify DEPTH_CLEAR_VALUE entry in table 6.21 (Framebuffer Control) p. 280

Get Value	Type	Get Command	Init Value	Description	Sec	Attribute
DEPTH_CLEAR_VALUE	R	GetFloatv	1	Depth buffer clear value	4.2.3	depth-buffer

Add DEPTH_BUFFER_FLOAT_MODE entry to table 6.32 (Implementation Dependent Values) p. 293

Get Value	Type	Get Command	Init Value	Description	Sec	Attribute
DEPTH_BUFFER_FLOAT_MODE	B	GetBooleanv	-	True if depth buffer uses a floating-point represnetation	4	-

Additions to Appendix A of the OpenGL 2.0 Specification (Invariance)

None.

Additions to the AGL/GLX/WGL Specifications

None.

GLX Protocol

None.

Dependencies on EXT_depth_bounds_test:

Modify the definition of DepthBoundsEXT in section 4.1.x Depth Bounds Test.

Modify section 4.1.x (Depth Bounds Test)

(modify first paragraph) ...These values are set with

```
    void DepthBoundsEXT(clampd zmin, clampd zmax);
    void DepthBoundsdNV(double zmin, double zmax);
```

The paramerters to DepthBoundsEXT are clamped to the range [0, 1]. No clamping is applied to the parameters of DepthBoundsdNV. Each of zmin and zmax are subject to clamping to the range of the depth buffer at the time the depth bounds test is applied. For fixed-point depth buffers, the applied zmin and zmax are clamped to [0, 1]. For floating-point depth buffers, the applied zmin and zmax are unmodified. If zmin <= Zpixel <= zmax, then the depth bounds test passes. Otherwise, the test fails and the fragment is discarded. The test is enabled or disabled using Enable or Disable using the constant DEPTH_BOUNDS_TEST_EXT. When disabled, it is as if the depth bounds test always passes. If zmin is greater than zmax, then the error INVALID_VALUE is generated. The state required consists of two floating-point values and a bit indicating whether the test is enabled or disabled. In the initial state, zmin and zmax are set to 0.0 and 1.0 respectively; and the depth bounds test is disabled.

Errors

Modify the following error in the EXT_packed_depth_stencil specification by adding mention of FLOAT_32_UNSIGNED_INT_24_8_REV_NV:

The error INVALID_ENUM is generated if DrawPixels or ReadPixels is called where format is DEPTH_STENCIL_EXT and type is not UNSIGNED_INT_24_8_EXT, or FLOAT_32_UNSIGNED_INT_24_8_REV_NV.

Modify the following error in the EXT_packed_depth_stencil specification by adding mention of FLOAT_32_UNSIGNED_INT_24_8_REV_NV:

The error INVALID_OPERATION is generated if DrawPixels or ReadPixels is called where type is UNSIGNED_INT_24_8_EXT, or FLOAT_32_UNSIGNED_INT_24_8_REV_NV and format is not DEPTH_STENCIL_EXT.

Add the following error to the NV_copy_depth_to_color specification:

The error INVALID_OPERATION is generated if CopyPixels is called where type is DEPTH_STENCIL_TO_RGBA_NV or DEPTH_STENCL_TO_BGRA_NV and the depth buffer uses a floating point representation.

New State

None.

Issues

1. *Should this extension expose floating-point depth buffers through WGL/GLX "pixel formats?"*

   RESOLVED: No. The WGL_EXT_depth_float extension already provides a mechanism for requesting a floating-point depth buffer.

2. *How does an application access the full range of a floating-point depth buffer?*

   RESOLVED: New functions have been introduced that set existing GL state without clamping to the range [0, 1]. These functions are DepthRangedNV, ClearDepthdNV, and DepthBoundsdNV.

3. *Should we add a new state query to determine if the depth buffer is using a floating-point representation?*

   RESOLVED: Yes. An application can query DEPTH_FLOAT_MODE_NV to see if the depth buffer is using a floating-point representation.

4. *How does polygon offset work with floating-point depth buffers?*

   RESOLVED: The third paragraph of section 3.5.5 (Depth Offset) describes the minimum resolvable difference r as "the smallest difference in window coordinate z values that is guaranteed to remain distinct throughout polygon rasterization and in the depth buffer." The polygon offset value o is computed as a function of r. The minimum resolvable difference r makes sense for fixed-point depth values, and even floating-point depth values in the range [-1, 1]. For unclamped floating-point depth values, there is no constant minimum resolvable difference -- the minimum difference necessary to change the mantissa of a floating-point value by one bit depends on the exponent of the value being offset. To remedy this problem, the minimum resolvable difference is defined to be relative to the range of depth values for the given primitive when the depth buffer is floating-point.

5. *How does NV_copy_depth_to_color work with floating-point depth values?*

   RESOLVED: It isn't clear that there is any usefulness to copying the data for 32-bit floating-point depth values to a fixed-point color buffer. It is even less clear how copying packed data from a FLOAT_32_UNSIGNED_24_8_NV depth/stencil buffer to a fixed-point color buffer would be useful or even how it should be implemented. An error should be generated if CopyPixels is called where <type> is DEPTH_STENCIL_TO_RGBA_NV or DEPTH_STENCIL_TO_BGRA and the depth buffer uses a floating-point representation.

6. *Other OpenGL hardware implementations may be capable of supporting floating-point depth buffers. Why is this an NV extension?*

   RESOLVED: When rendering to floating-point depth buffers, we expect that other implementations may only be capable of supporting Z values in the range [0,1]. For such implementations, floating-point Z buffers do not improve the range of Z values supported, but do offer

    increased precision than conventional 24-bit fixed-point Z buffers,
    particularly around zero.

    This extension was initially proposed as an EXT, but we have changed
    it to an NV extension in the expectation that an EXT may be offered at
    some point in the not-too-distant future. We expect that the EXT
    could be supported by a larger range of vendors. NVIDIA would
    continue to support both extensions, where the NV extension could be
    thought of as taking the capability of the EXT version and extending
    it to support Z values outside the range [0,1].

Revision History

None

Name

    NV_depth_clamp

Name Strings

    GL_NV_depth_clamp

Notice

Status

    Implemented

Version

    Last Modified Date: $Date: 2002/02/13 $
    NVIDIA Revision: $Revision: #1 $

Number

    260

Dependencies

Written based on the wording of the OpenGL 1.2.1 specification.

Overview

Conventional OpenGL clips geometric primitives to a clip volume with six faces, two of which are the near and far clip planes. Clipping to the near and far planes of the clip volume ensures that interpolated depth values (after the depth range transform) must be in the [0,1] range.

In some rendering applications such as shadow volumes, it is useful to allow line and polygon primitives to be rasterized without clipping the primitive to the near or far clip volume planes (side clip volume planes clip normally). Without the near and far clip planes, rasterization (pixel coverage determination) in X and Y can proceed normally if we ignore the near and far clip planes. The one major issue is that fragments of a primitive may extend beyond the conventional window space depth range for depth values (typically the range [0,1]). Rather than discarding fragments that defy the window space depth range (effectively what near and far plane clipping accomplish), the depth values can be clamped to the current depth range.

This extension provides exactly such functionality. This functionality is useful to obviate the need for near plane capping of stenciled shadow volumes. The functionality may also be useful for rendering geometry "beyond" the far plane if an alternative algorithm (rather than depth testing) for hidden surface removal is applied to such geometry (specifically, the painter's algorithm). Similar situations at the near clip plane can be avoided at the near clip plane where apparently solid objects can be "seen through" if they intersect the near clip plane.

Issues

Another way to specify this functionality is to describe it in terms of generating the equivalent capping geometry that would need to be drawn at the near or far clip plane to have the same effect as not clipping to the near and far clip planes and clamping interpolated depth values outside the window-space depth range. Should the functionality be described this way?

RESOLUTION: No. Describing the functionality as capping is fairly involved. Eliminating far and near plane clipping and clamping interpolated depth values to the depth range is much simpler to specify.

Should depth clamping affect points or just line and polygon geometric primitives?

RESOLUTION: All geometric primitives are affected by depth clamping.

In the case of points, if you render a point "in front of" the near clip plane, it should be rendered with the zw value min(n,f) where n and f are the near and far depth range values if depth clamping is enabled. Similarly, a point "behind" the far clip plane should be rendered with the zw value max(n,f).

How should the setting of the raster position function when depth clamping is enabled?

RESOLUTION: When setting the raster position, clamp the raster position zw to the range [min(n,f),max(n,f)] where n and f are the near and far depth range values.

This rule ensures that the raster position zw will never be outside the [0,1] range (because n and far are clamped to the [0,1] range). We specify the raster position to be updated this way because otherwise a raster position zw could be specified outside the [0,1] range when depth clamping is enabled, but then if depth clamping is subsequently disabled, that out-of-range raster position zw could not be written to the depth buffer.

This semantic can make for some unexpected semantics that are described here. Say that depth clamping is enabled and the raster position is set to point behind the far clip plane such that the pre-clamped zw is 2.5. Because depth clamping is enabled the raster position zw is clamped to the current near and far depth range values. Say these values are 0.1 and 0.9. So 2.5 is clamped to 0.9.

Now consider what happens if a bitmap (or image rectangle) is rendered with depth testing enabled under various situations. If depth clamping remains enabled and the depth range is unchanged, the bitmap fragments are generated with a zw of 0.9.

However, if depth range is subsequently set to 0.2 and 0.8 and depth clamping is enabled, the bitmap fragments will have their zw depth component clamped to 0.8. But if the depth range was changed to 0.2 and 0.8 but depth range clamped is disabled, the bitmap fragments will have a 0.9 zw depth component since then the depth clamping is then not applied.

What push/pop attrib bits should affect the depth clamp enable?

RESOLUTION: GL_ENABLE_BIT and GL_TRANSFORM_BIT.

How does depth clamping interact with depth replace operations (say from NV_texture_shader)?

RESOLUTION: The depth clamp operation occurs as part of the depth test so depth clamping occurs AFTER any depth replace operation in the pipeline. A depth replace operation can reassign the fragment's zw, but depth clamping if enabled will subsequently clamp this new zw.

Does depth clamping affect read/draw/copy pixels operations involving depth component pixels?

RESOLUTION: No.

Does depth clamping occur after polygon offset?

RESOLUTION: Yes. Depth clamping occurs immediately before the depth test.

Can fragments with wc<=0 be generated when this extension is supported?

RESOLUTION: No. The core OpenGL specification (section 2.11) is worded to allow the possibility of generating fragments where wc<=0. These should never be generated when this extension is supported.

New Procedures and Functions

None

New Tokens

Accepted by the <cap> parameter of Enable, Disable, and IsEnabled, and by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

```
    DEPTH_CLAMP_NV                              0x864F
```

Additions to Chapter 2 of the OpenGL 1.2.1 Specification (OpenGL Operation)

-- Section 2.11 "Clipping"

Add to the end of the 3rd paragraph:

"Depth clamping is enabled with the generic Enable command and disabled with the Disable command. The value of the argument to either command is DEPTH_CLAMP_NV. If depth clamping is enabled, the

    "-wc <= zc <= wc" plane equation are ignored by video volume clipping
    (effectively, there is no near or far plane clipping)."

    Change the 8th paragraph to indicate that only wc>0 fragments should
    be generated rather than even allowing the posibility that wc<=0
    fragments may be generated:

    "A line segment or polygon whose vertices have wc values of differing
    signs may generate multiple connected components after clipping.
    GL implementations are not required to handle this situation.
    That is, only the portion of the primitive that lies in the region
    of wc>0 should be produced by clipping."

-- Section 2.12 "Current Raster Position"

    Add to the end of the 4th paragraph:

    "If depth clamping (see section 2.11) is enabled, then raster position
    zw is first clamped as follows. If the raster postition's wc>0,
    then zw is clamped the range [min(n,f),max(n,f)] where n and f are
    the current near and far depth range values (see section 2.10.1)."

Additions to Chapter 3 of the OpenGL 1.2.1 Specification (Rasterization)

    None

Additions to Chapter 4 of the OpenGL 1.2.1 Specification (Per-Fragment Operations and the Framebuffer)

-- Section 4.1.5 "Depth buffer test"

    Add to the end of the 2nd paragraph:

    "If depth clamping (see section 2.11) is enabled, before the incoming
    fragment's zw is compared, zw must first be clamped as follows: If the
    fragment's wc>0, then zw is clamped to the range [min(n,f),max(n,f)]
    where n and f are the current near and far depth range values (see
    section 2.10.1)."

Additions to Chapter 5 of the OpenGL 1.2.1 Specification (Special Functions)

    None

Additions to Chapter 6 of the OpenGL 1.2.1 Specification (State and State Requests)

    None

Additions to the AGL/GLX/WGL Specifications

    None

GLX Protocol

    None

Errors

None

New State

(table 6.7)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
DEPTH_CLAMP_NV	B	IsEnabled	False	Depth clamping on/off	2.10.2	transform/enable

New Implementation Dependent State

None

Revision History

None

Name

    NV_evaluators

Name Strings

    GL_NV_evaluators

Notice

IP Status

    NVIDIA Proprietary.

Version

    NVIDIA Date: April 13, 2001
    $Id: //sw/main/docs/OpenGL/specs/GL_NV_evaluators.txt#2 $

Number

    225

Dependencies

    Written based on the wording of the OpenGL 1.2.1 specification.

    Assumes support for the ARB_multitexture extension.

    NV_vertex_program affects the definition of this extension.

Overview

    OpenGL evaluators provide applications with the capability to
    specify polynomial or rational curves and surfaces using control
    points relative to the Bezier basis. The curves and surfaces are
    then drawn by evaluating the polynomials provided at various values
    for the u parameter of a curve or the (u,v) parameters of a surface.
    A tensor product formulation is used for the surfaces.

    For various historical reasons, evaluators have not been
    particularly popular as an interface for drawing curves and surfaces.
    This extension proposes a new interface for surfaces that provides a
    number of significant enhancements to the functionality provided by
    the original OpenGL evaluators.

    Many implementations never optimized evaluators, so applications
    often implemented their own algorithms instead. This extension
    relaxes some restrictions that make it difficult to optimize
    evaluators.

    Also, new vertex attributes have been added to OpenGL through
    extensions, including multiple sets of texture coordinates, a
    secondary color, a fog coordinate, a vertex weight, and others.
    The extensions which added these vertex attributes never bothered

to update the functionality of evaluators, since they were used so little in the first place. In turn, evaluators have become more and more out of date, making it even less likely that developers will want to use them. Most of the attributes are not a big loss, but support for multiple sets of texture coordinates would be absolutely essential to developers considering the use of evaluators.

OpenGL evaluators only support rectangular patches, not triangular patches. Although triangular patches can be converted into rectangular patches, direct support for triangular patches is likely to be more efficient.

The tessellation algorithm used is too inflexible for most purposes; only the number of rows and columns can be specified. Adjacent patches must then have identical numbers of rows and columns, or severe cracking will occur. Ideally, a number of subdivisions could be specified for all four sides of a rectangular patch and for all three of a triangular patch. This extension goes one step further and allows those numbers to be specified in floating-point, providing a mechanism for smoothly changing the level of detail of the surface.

Meshes evaluated with EvalMesh are required to match up exactly with equivalent meshes evaluated with EvalCoord or EvalPoint. This makes it difficult or impossible to use optimizations such as forward differencing.

Finally, little attention is given to some of the difficult problems that can arise when multiple patches are drawn. Depending on the way evaluators are implemented, and depending on the orientation of edges, numerical accuracy problems can cause cracks to appear between patches with the same boundary control points. This extension makes guarantees that an edge shared between two patches will match up exactly under certain conditions.

Issues

* *Should one-dimensional evaluators be supported?*

  RESOLVED: No. This extension is intended for surfaces only.

* *Should we support triangular patches?*

  RESOLVED: Yes. Otherwise, applications will have to convert them to rectangular patches themselves. We can do this more efficiently.

* *What domain should triangular patches be defined on?*

  RESOLVED: (0,0),(1,0),(0,1).

* *What memory layout should we use for triangular patch control points?*

  RESOLVED: Both a[i][j], where i+j <= n, and a packed format are supported.

* *Is it worth it to have "evaluator objects"?*

  RESOLVED: No. We will leave these out for now.

* *Should we support the original evaluators' ability to specify a map from an arbitrary (u1,v1) to an arbitrary (u2,v2)?*

  RESOLVED: No. Maps will always extend from (0,0) to (1,1) and will always be evaluated from (0,0) to (1,1).

* *Should the new interface support an EvalCoord-like syntax?*

  RESOLVED: No. We are only interested in evaluating an entire mesh at once.

* *Should we support the "mode" parameter to the existing EvalMesh2, which allows the mesh to be tessellated in wireframe or as points?*

  RESOLVED: No. We will leave in the parameter and require that it be FILL, though, to leave room for a future extension.

* *Should there be a new interface to specify control points or should Map2{fd} be reused?*

  RESOLVED: A new interface. There are enough changes compared to the original evaluators that we can't reuse the old interface without causing more problems. For example, the target parameter of Map2{fd} is really a cross of target and index in MapControlPointsNV, and so it ends up creating an excessive number of enumerants.

* *How should grids be specified?*

  RESOLVED: A MapParameter command. This is better than a new MapGrid- style command because it can be extended to include new parameter types.

* *Should there be any rules about the order of generation of primitives within a single patch?*

  RESOLVED: No. The tessellation algorithm itself is not even specified, so it makes no sense to do this. Applications must not depend on the order in which the primitives are drawn.

* *Should the stride for MapControlPointsNV be specified in basic machine units (i.e. unsigned bytes) or in floats/doubles?*

  RESOLVED: ubytes. Most of the rest of OpenGL (vertex arrays, pixel path, etc.) uses ubytes; evaluators are actually inconsistent.

* *How much leeway should implementations be given to choose their own algorithm for tessellation?*

  RESOLVED: The integral tessellation scheme will require a specific tessellation of the boundary edges of the patch, but the interior tessellation is implementation-specific. The fractional tessellation scheme will only require a minimum number of segments along each edge. In either case, a minimum number of triangles for the entire patch is specified.

* *Should there be rules to ensure that the triangles will be oriented in a consistent fashion?*

  RESOLVED: Yes. This is essential for features such as backface culling to work properly. The rule given ensures that the orientation will be identical to the orientation used for the original evaluators.

* *Should there be a separate MAX_EVAL_ORDER for rational surfaces?*

  RESOLVED: Yes. Rational surfaces require additional calculation to be done by the implementation, especially if AUTO_NORMAL is enabled. Furthermore, the most useful rational surfaces are of low order. For example, all the conic sections are quadratic rational surfaces.

* *Should there be enables similar to AUTO_NORMAL that generate partials of U (tangents), partials of V, and/or binormals?*

  RESOLVED: No. The application is responsible for configuring the evaluators appropriately.

  The auto normal functionality is supported because it is fairly complicated and was already a core part of OpenGL for evaluators. Plus there is already a "normal" vertex attribute for it to automatically generate.

  The partials of U and partials of V are fairly straightforward to evaluate (just take the derivative of the bivariate polynomial in terms of either U or V) plus there is not a particular vertex attribute associated with each of these.

New Procedures and Functions

```
    void MapControlPointsNV(enum target, uint index, enum type,
                            sizei ustride, sizei vstride,
                            int uorder, int vorder,
                            boolean packed,
                            const void *points)

    void MapParameterivNV(enum target, enum pname, const int *params)
    void MapParameterfvNV(enum target, enum pname, const float *params)

    void GetMapControlPointsNV(enum target, uint index, enum type,
                               sizei ustride, sizei vstride,
                               boolean packed, void *points)

    void GetMapParameterivNV(enum target, enum pname, int *params)
    void GetMapParameterfvNV(enum target, enum pname, float *params)
    void GetMapAttribParameterivNV(enum target, uint index, enum pname,
                                   int *params)
    void GetMapAttribParameterfvNV(enum target, uint index, enum pname,
                                   float *params)
    void EvalMapsNV(enum target, enum mode)
```

New Tokens

```
    Accepted by the <target> parameter of MapControlPointsNV,
    MapParameter[if]vNV, GetMapControlPointsNV, GetMapParameter[if]vNV,
    GetMapAttribParameter[if]vNV, and EvalMapsNV:

        EVAL_2D_NV                              0x86C0
        EVAL_TRIANGULAR_2D_NV                   0x86C1

    Accepted by the <pname> parameter of MapParameter[if]vNV and
    GetMapParameter[if]vNV:

        MAP_TESSELLATION_NV                     0x86C2

    Accepted by the <pname> parameter of GetMapAttribParameter[if]vNV:

        MAP_ATTRIB_U_ORDER_NV                   0x86C3
        MAP_ATTRIB_V_ORDER_NV                   0x86C4

    Accepted by the <cap> parameter of Disable, Enable, and IsEnabled,
    and by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv,
    and GetDoublev:

        EVAL_FRACTIONAL_TESSELLATION_NV         0x86C5

        EVAL_VERTEX_ATTRIB0_NV                  0x86C6
        EVAL_VERTEX_ATTRIB1_NV                  0x86C7
        EVAL_VERTEX_ATTRIB2_NV                  0x86C8
        EVAL_VERTEX_ATTRIB3_NV                  0x86C9
        EVAL_VERTEX_ATTRIB4_NV                  0x86CA
        EVAL_VERTEX_ATTRIB5_NV                  0x86CB
        EVAL_VERTEX_ATTRIB6_NV                  0x86CC
        EVAL_VERTEX_ATTRIB7_NV                  0x86CD
        EVAL_VERTEX_ATTRIB8_NV                  0x86CE
        EVAL_VERTEX_ATTRIB9_NV                  0x86CF
        EVAL_VERTEX_ATTRIB10_NV                 0x86D0
        EVAL_VERTEX_ATTRIB11_NV                 0x86D1
        EVAL_VERTEX_ATTRIB12_NV                 0x86D2
        EVAL_VERTEX_ATTRIB13_NV                 0x86D3
        EVAL_VERTEX_ATTRIB14_NV                 0x86D4
        EVAL_VERTEX_ATTRIB15_NV                 0x86D5

    Accepted by the <pname> parameter of GetBooleanv, GetIntegerv,
    GetFloatv, and GetDoublev:

        MAX_MAP_TESSELLATION_NV                 0x86D6
        MAX_RATIONAL_EVAL_ORDER_NV              0x86D7
```

Additions to Chapter 2 of the 1.2 Specification (OpenGL Operation)

None.

Additions to Chapter 3 of the 1.2 Specification (Rasterization)

None.

Additions to Chapter 4 of the 1.2 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the 1.2 Specification (Special Functions)

-- NEW Section 5.7 "General Evaluators"

"General evaluators are similar to evaluators in that they can be used to evaluate polynomial and rational mappings, but general evaluators have several new features that the original evaluators do not. First, they support triangular surfaces in addition to (quadrilateral) tensor product surfaces. Second, the tessellation can be varied continuously as well as in integral steps. Finally, general evaluators can evaluate all vertex attributes, not just the vertex, color, normal, and texture coordinates.

Several elements of the original evaluators have been removed in the general evaluators interface. The general evaluators always evaluate four components in parallel, whereas the original evaluators might evaluate between 1 and 4 (see the "k" column in Table 5.1 on page 165). The original evaluators can map on an arbitrary domain and can map grids on an arbitrary region, whereas the general evaluators only use the [0,1] range. Support for 1D evaluators, an EvalCoord-style interface, and the "mode" parameter of EvalMesh* has also been removed from the general evaluators.

The command

```
  void MapControlPointsNV(enum target, uint index, enum type,
                          sizei ustride, sizei vstride,
                          int uorder, int vorder, boolean packed,
                          const void *points);
```

specifies control points for a general evaluator map. target is the type of evaluator map and can be either EVAL_2D_NV or EVAL_TRIANGULAR_2D_NV. index is the number of the vertex attribute register the map will be used to evaluate for; these are the same indices used in the GL_NV_vertex_program extension. Table X.1 shows the relationship between these indices and the conventional per-vertex attributes for implementations that do not support GL_NV_vertex_program.

Vertex Attribute Register Number	Conventional Per-vertex Parameter	Conventional Per-vertex Parameter Command	Component Mapping
0	vertex position	Vertex	x,y,z,w
1	vertex weights	VertexWeightEXT	w,0,0,1
2	normal	Normal	x,y,z,1
3	primary color	Color	r,g,b,a
4	secondary color	SecondaryColorEXT	r,g,b,1
5	fog coordinate	FogCoordEXT	fc,0,0,1
6	-	-	-
7	-	-	-
8	texture coord 0	MultiTexCoordARB(GL_TEXTURE0_ARB, ...)	s,t,r,q
9	texture coord 1	MultiTexCoordARB(GL_TEXTURE1_ARB, ...)	s,t,r,q
10	texture coord 2	MultiTexCoordARB(GL_TEXTURE2_ARB, ...)	s,t,r,q
11	texture coord 3	MultiTexCoordARB(GL_TEXTURE3_ARB, ...)	s,t,r,q
12	texture coord 4	MultiTexCoordARB(GL_TEXTURE4_ARB, ...)	s,t,r,q
13	texture coord 5	MultiTexCoordARB(GL_TEXTURE5_ARB, ...)	s,t,r,q
14	texture coord 6	MultiTexCoordARB(GL_TEXTURE6_ARB, ...)	s,t,r,q
15	texture coord 7	MultiTexCoordARB(GL_TEXTURE7_ARB, ...)	s,t,r,q

Table X.1: Aliasing of vertex attributes with conventional per-vertex parameters.

type is either FLOAT or DOUBLE. ustride and vstride are the numbers of basic machine units (typically unsigned bytes) between control points in the u and v directions. uorder and vorder have the same meaning they do in the Map2{fd} command. The error INVALID_VALUE is generated if either uorder or vorder is less than one or greater than MAX_EVAL_ORDER. The error INVALID_OPERATION is generated if target is EVAL_TRIANGULAR_2D_NV and uorder is not equal to vorder.

points is a pointer to an array of control points. If target is EVAL_2D_NV, there are uorder*vorder control points in the array, and if it is EVAL_TRIANGULAR_2D_NV, there are uorder*(uorder+1)/2 points in the array. If packed is FALSE, control point i,j is located

```
    (ustride)i + (vstride)j
```

basic machine units from points. If target is EVAL_2D_NV, i ranges from 0 to uorder-1, and j ranges from 0 to vorder-1. If target is EVAL_TRIANGULAR_2D_NV, i and j range from 0 to uorder-1, and i+j must be less than or equal to uorder-1.

If packed is TRUE and target is EVAL_2D_NV, control point i,j is located

```
    (ustride)(j*uorder + i)
```

basic machine units from points. If packed is TRUE and target is EVAL_TRIANGULAR_2D_NV, control point i,j is located

```
    (ustride)(j*uorder + i - j*(j-1)/2)
    basic machine units from points.

    The error INVALID_OPERATION is generated if index is 0, one of the
    control points' fourth components is not equal to 1, and either uorder
    of vorder is greater than MAX_RATIONAL_EVAL_ORDER_NV.

    The evaluation of a 2D tensor product map is performed in the same
    way as for the original evaluators.  The exact coordinates produced
    by the original evaluators may differ from those produced by the
    general evaluators, since different algorithms may be used.

    A triangular map may be evaluated as follows.  Let Ri,j be the
    4-component vector for control point i,j and n be the degree of the
    patch (i.e.  uorder-1).  Then:

                   ---
                   \  (n) (n-i)  i  j        n-i-j
        p_t(u,v) = /  (i) ( j ) u  v  (1-u-v)      Ri,j
                   ---
                i,j >= 0
                i+j <= n

    evaluates the point p_t(u,v) on the triangular patch at parameter
    values (u,v).  (The notation on the left indicates "n choose i" and
    "n minus i choose j", i.e., binomial coefficients.)

    The evaluation of any particular attribute can be enabled or disabled
    with Enable and Disable using one of the EVAL_VERTEX_ATTRIBi_NV
    constants.

    If AUTO_NORMAL is enabled (see section 5.1), analytically computed
    normals are evaluated as well.  The formula for the normal is the same
    as the one in section 5.1, except that the magnitude of the normals is
    undefined.  These normals should be renormalized by enabling NORMALIZE,
    or by normalizing them in a vertex program.  The w of the normal
    vertex attribute will always be 1.

    The commands

      void MapParameter{if}vNV(enum target, enum pname, T params);

    can be used to specify the level of tessellation to evaluate,
    where target is EVAL_2D_NV or EVAL_TRIANGULAR_2D_NV and pname is
    MAP_TESSELLATION_NV.  If target is EVAL_2D_NV, params contains the
    four values [nu0,nu1,nv0,nv1], and if it is EVAL_TRIANGULAR_2D_NV,
    params contains the three values [n1,n2,n3].  The state for each
    target is independent of the other.  These values are clamped to
    the range [1.0, MAX_MAP_TESSELLATION_NV].

    The use of a fractional tessellation algorithm can be
    enabled or disabled with Enable and Disable using the
    EVAL_FRACTIONAL_TESSELLATION_NV constant.  The fractional tessellation
    algorithm allows the tessellation to smoothly morph without popping
    as the tessellation parameters are varied by small amounts.
```

The command

```
  void EvalMapsNV(enum target, enum mode);
```

evaluates the currently enabled maps. target is either EVAL_2D_NV or EVAL_TRIANGULAR_2D and specifies which set of maps to evaluate. mode must be FILL. If EVAL_VERTEX_ATTRIB0_NV is not enabled, the error INVALID_OPERATION results.

If EVAL_FRACTIONAL_TESSELLATION_NV is disabled, tensor product maps are evaluated such that the boundaries of the mesh are divided into ceil(nu0) segments on the edge from (0,0) to (1,0), ceil(nu1) segments on the edge from (0,1) to (1,1), ceil(nv0) segments on the edge from (0,0) to (0,1), and ceil(nv1) segments on the edge from (1,0) to (1,1). These segments must be evaluated at equal spacings in (u,v) parameter space.

Triangular maps are evaluated such that the boundary of the mesh from (0,0) to (1,0) has ceil(n1) equally-spaced segments, the boundary from (1,0) to (0,1) has ceil(n2) equally-spaced segments, and the boundary from (0,1) to (0,0) has ceil(n3) equally-spaced segments.

If EVAL_FRACTIONAL_TESSELLATION_NV is enabled, each edge must be tessellated with no fewer the number of segments that would be used in the non- fractional case for any values of the tessellation parameters. Furthermore, the tessellation of each edge must vary smoothly with the parameters; that is, a small change in any or all of the parameters must cause a small change in the tessellation. Whenever a new vertex is introduced into the tessellation, it must be coincident with another vertex, and whenever a vertex is removed, it must have been coincident with a different vertex. The parameter-space position of any vertex must be a continuous function of the tessellation parameters.

The same minimum triangle requirements apply to fractional tessellations as to integral tessellations.

A tensor product patch must always be tessellated with no fewer than

```
    2 * ceil((nu0+nu1)/2) * ceil((nv0+nv1)/2)
```

triangles in total.

A triangular patch must always be tessellated with no fewer than

```
    ceil((n1+n2+n3)/3)^2
```

triangles in total.

If a triangle is formed by evaluating the maps at the three coordinates (u1,v1), (u2,v2), and (u3,v3), it must be true that

```
    (u3-u1)*(v2-v1) - (u2-u1)*(v3-v1) >= 0
```

to ensure that all triangles in a patch have a consistent orientation.

The current value of any vertex attribute for which the evaluation of a map is enabled becomes undefined after an EvalMapsNV command. If AUTO_NORMAL is enabled, the current normal becomes undefined as well.

If AUTO_NORMAL is enabled, the analytically computed normals take precedence over the currently enabled map for vertex attribute 2 (the normal).

To prevent cracks, certain rules must be established for performing the evaluations. The goal of these rules is to ensure that no matter what order control points are specified in and what the tessellation parameters are, so long as the control points on any edge exactly match the control points of an adjacent edge, and so long as the subdivision parameter for that edge is the same for the adjacent patch, there will be no cracking artifacts between the two patches. These requirements are completely independent of numerical precision. In particular, we will require that these shared vertices' positions be equal. Furthermore, there must be no cracking inside the geometry of any patch, and normals must not change in a discontinuous fashion so that there are no discontinuities in lighting or other effects that use the normal.

Let two patches share an edge of equal order (the order of an edge is the order of the patch in that direction for a tensor product patch, or the order of the patch for a triangular patch). Then this edge is said to be consistent if all the vertex control points (vertex attribute 0) are identical on each edge (although they may be specified in the opposite direction, or even in a different coordinate; one may an edge in the u direction, and one may be an edge in the v direction).

If an edge is consistent, and if each of the two patches are tessellated with identical tessellation parameters for that edge, then the vertex coordinates given to vertex processing must be exactly equal for each of the vertices.

The vertex coordinates given to vertex processing for the corner vertices of any patch must be exactly equal to the values of the corner control points of that patch, regardless of the patch's order, type, tessellation parameters, the state of the AUTO_NORMAL or EVAL_FRACTIONAL_TESSELLATION_NV enables, the control points, order, or enable of any other associated map, or any other OpenGL state.

The vertex coordinates and normals given to vertex processing for any vertex of a patch must be exactly equal each time that vertex is evaluated during the tessellation of a patch. Since each vertex is shared between several triangles in the patch, any variation in these coordinates and normals would result in cracks or lighting discontinuities.

The state required for the general evaluators consists of a bit indicating whether fractional tessellation is enabled or disabled, 16 bits indicating whether the evaluation of each vertex attribute is enabled or disabled, four floating-point map tessellation values for tensor product patches, three floating-point map tessellation values for triangular patches, and a map specification for a tensor product patch and a triangular patch for each vertex attribute. A map specification consists of two integers indicating the order of the map in u and v and a two-dimensional array of vectors of four floating-point values containing the control points for that map. The initial state of fractional tessellation is disabled. The initial state of evaluation of vertex attribute 0 is enabled, and the initial state of evaluation for any other vertex attribute is disabled. The initial value of the tessellation parameters is 1.0. The initial order of each map specification is an order of 1 in both u and v and a single control point of [0,0,0,1]."

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

-- NEW Section 6.1.13 "General Evaluator Queries"

"The commands

```
  void GetMapParameterivNV(enum target, enum pname, int *params);
  void GetMapParameterfvNV(enum target, enum pname, float *params);
```

obtain the parameters for a map target. target may be one of EVAL_2D_NV or EVAL_TRIANGULAR_2D_NV. pname must be MAP_TESSELLATION_NV. The map tessellation is placed in params.

The commands

```
  void GetMapAttribParameterivNV(enum target, uint index, enum pname,
                                 int *params);
  void GetMapAttribParameterfvNV(enum target, uint index, enum pname,
                                 float *params);
```

obtain parameters for a single map. target may be one of EVAL_2D_NV or EVAL_TRIANGULAR_2D_NV. index is the number of the vertex attribute register the map is used for evaluating. If pname is MAP_ATTRIB_U_ORDER_NV, the u order of the map is placed in params. If pname is MAP_ATTRIB_V_ORDER_NV, the v order of the map is placed in params.

The command

```
  void GetMapControlPointsNV(enum target, uint index, enum type,
                             sizei ustride, sizei vstride, boolean packed,
                             void *points);
```

obtains the control points of a map. target may be one of EVAL_2D_NV or EVAL_TRIANGULAR_2D_NV. index is the number of the vertex attribute register the map is used for evaluating. type is either FLOAT or DOUBLE. ustride and vstride are the numbers of basic machine units (typically unsigned bytes) between control points in the u and v directions. points is a pointer to an array where the control points are to be written. If target is EVAL_2D_NV, there are uorder*vorder control points in the array, and if it is EVAL_TRIANGULAR_2D_NV, there are uorder*(uorder+1)/2 points in the array. If packed is FALSE, control point i,j is located

```
    (ustride)i + (vstride)j
```

basic machine units from points. If packed is TRUE and target is

```
    EVAL_2D_NV, control point i,j is located

        (ustride)(j*uorder + i)

    basic machine units from points.  If packed is TRUE and target is
    EVAL_TRIANGULAR_2D_NV, control point i,j is located

        (ustride)(j*uorder + i - j*(j-1)/2)

    basic machine units from points.  If target is EVAL_2D_NV, i ranges
    from 0 to uorder-1, and j ranges from 0 to vorder-1.  If target is
    EVAL_TRIANGULAR_2D_NV, i and j range from 0 to uorder-1, and i+j
    must be less than or equal to uorder-1."
```

Additions to the GLX Specification

```
    Nine new GL commands are added.

    The following three rendering commands are sent to the sever
    as part of a glXRender request:

        MapParameterivNV
            2           12+4*n                rendering command length
            2           ????                  rendering command opcode
            4           ENUM                  target
            4           ENUM                  pname
                        0x86C2                GL_MAP_TESSELLATION_NV
                                      n=3     if (target == GL_EVAL_TRIANGULAR_2D_NV)
                                      n=4     if (target == GL_EVAL_2D_NV)
                        else          n=0     command is erroneous
            4*n         LISTofINT32           params

        MapParameterfvNV
            2           12+4*n                rendering command length
            2           ????                  rendering command opcode
            4           ENUM                  target
            4           ENUM                  pname
                        0x86C2                GL_MAP_TESSELLATION_NV
                                      n=3     if (target == GL_EVAL_TRIANGULAR_2D_NV)
                                      n=4     if (target == GL_EVAL_2D_NV)
                        else          n=0     command is erroneous
            4*n         LISTofFLOAT32         params

        EvalMapsNV
            2           12                    rendering command length
            2           ????                  rendering command opcode
            4           ENUM                  target
            4           ENUM                  mode

    The following rendering command is potentially large and can be sent
    in a glXRender or glXRenderLarge request:

        MapControlPointsNV
            2           24+m                rendering command length
            2           ????                rendering command opcode
            4           ENUM                target
            4           CARD32              index
            4           CARD32              type
            4           INT32               uorder
            4           INT32               vorder
            m           (see below)         points

          Determine m from the table below; n depends on the target.  If the
          target is GL_EVAL_2D_NV, then n = uorder*vorder.  If the target
          is GL_EVAL_TRIANGULAR_2D_NV, then n = uorder * (uorder+1)/2.
          The points data is packed such that when unpacked by the server,
          the value of ustride is 16 for GL_FLOAT typed data and 32 for
          GL_DOUBLE typed data.

           type       encoding of type  type of lists  m (bytes)
           ---------  ----------------  -------------  ---------
           GL_FLOAT   0x1406            LISTofFLOAT32  n*4
           GL_DOUBLE  0x140A            LISTofFLOAT64  n*8

          If the command is encoded in a glXRenderLarge request, the command
          opcode and command length fields above are expanded to 4 bytes each:

            4           28+m               rendering command length
            4           ????               rendering command opcode
```

The remaining five commands are non-rendering commands. These commands are sent separately (i.e., not as part of a glXRender or glXRenderLarge request), using the glXVendorPrivateWithReply request:

```
    GetMapParameterivNV
        1           CARD8           opcode (X assigned)
        1           17              GLX opcode (glXVendorPrivateWithReply)
        2           5               request length
        4           ????            vendor specific opcode
        4           GLX_CONTEXT_TAG context tag
        4           ENUM            target
        4           ENUM            pname
      =>
        1           1               reply
        1                           unused
        2           CARD16          sequence number
        4           m               reply length, m=(n==1?0:n)
        4                           unused
        4           CARD32          n

        if (n=1) this follows:

        4           INT32           params
        12                          unused

        otherwise this follows:

        16                          unused
        n*4         LISTofINT32     params

    GetMapParameterfvNV
        1           CARD8           opcode (X assigned)
        1           17              GLX opcode (glXVendorPrivateWithReply)
        2           5               request length
        4           ????            vendor specific opcode
        4           GLX_CONTEXT_TAG context tag
        4           ENUM            target
        4           ENUM            pname
      =>
        1           1               reply
        1                           unused
        2           CARD16          sequence number
        4           m               reply length, m=(n==1?0:n)
        4                           unused
        4           CARD32          n

        if (n=1) this follows:

        4           FLOAT32         params
        12                          unused

        otherwise this follows:

        16                          unused
        n*4         LISTofFLOAT32   params
    GetMapAttribParameterivNV
        1           CARD8           opcode (X assigned)
        1           17              GLX opcode (glXVendorPrivateWithReply)
        2           6               request length
        4           ????            vendor specific opcode
        4           GLX_CONTEXT_TAG context tag
        4           ENUM            target
        4           CARD32          index
        4           ENUM            pname
      =>
        1           1               reply
        1                           unused
        2           CARD16          sequence number
        4           m               reply length, m=(n==1?0:n)
        4                           unused
        4           CARD32          n

        if (n=1) this follows:

        4           INT32           params
        12                          unused

        otherwise this follows:

        16                          unused
        n*4         LISTofINT32     params

    GetMapAttribParameterfvNV
        1           CARD8           opcode (X assigned)
        1           17              GLX opcode (glXVendorPrivateWithReply)
        2           6               request length
        4           ????            vendor specific opcode
        4           GLX_CONTEXT_TAG context tag
        4           ENUM            target
        4           CARD32          index
        4           ENUM            pname
      =>
        1           1               reply
        1                           unused
        2           CARD16          sequence number
        4           m               reply length, m=(n==1?0:n)
        4                           unused
        4           CARD32          n

        if (n=1) this follows:

        4           FLOAT32         params
        12                          unused

        otherwise this follows:

        16                          unused
        n*4         LISTofFLOAT32   params
    GetMapControlPointsNV
        1           CARD8           opcode (X assigned)
        1           17              GLX opcode (glXVendorPrivateWithReply)
        2           6               request length
        4           ????            vendor specific opcode
        4           GLX_CONTEXT_TAG context tag
        4           ENUM            target
        4           CARD32          index
        4           ENUM            type
      =>
        1           1               reply
        1                           unused
        2           CARD16          sequence number
        4           m               reply length, m
        4                           unused
        4           CARD32          uorder
        4           CARD32          vorder
        12                          unused

        if type == 0x1406 (GL_FLOAT) and target == 0x86C0
        (GL_EVAL_2D_NV), m = 4*uorder*vorder and the packed control
        points follow assuming ustride = 16

        m*4         LISTofFLOAT32   params

        if type == 0x140A (GL_DOUBLE) and target == 0x86C0
        (GL_EVAL_2D_NV), m = 4*uorder*vorder and the packed control
        points follow asssuming ustride = 32

        m*8         LISTofFLOAT64   params

        if type == 0x1406 (GL_FLOAT) and target == 0x86C1
        (GL_EVAL_TRIANGULAR_2D_NV), m = 4*uorder*(uorder+1)/2 and
        the packed control points follow assuming ustride = 16

        m*4         LISTofFLOAT32   params

        if type == 0x140A (GL_DOUBLE) and target == 0x86C1
        (GL_EVAL_TRIANGULAR_2D_NV), m = 4*uorder*(uorder+1)/2 and
        the packed control points follow asssuming ustride = 32

        m*8         LISTofFLOAT64   params

        otherwise m = 0 and nothing else follows.
```

Errors

The error INVALID_VALUE is generated if MapControlPointsNV, GetMapControlPointsNV, or GetMapAttribParameter{if}v is called where index is greater than 15.

The error INVALID_VALUE is generated if MapControlPointsNV or GetMapControlPointsNV is called where ustride or vstride is negative.

The error INVALID_VALUE is generated if MapControlPointsNV is called where uorder or vorder is less than one or greater than MAX_EVAL_ORDER.

The error INVALID_OPERATION is generated if MapControlPointsNV is called where target is EVAL_TRIANGULAR_2D_NV and uorder is not equal to vorder.

The error INVALID_OPERATION is generated if MapControlPointsNV is called where index is 0, one of the control points' fourth components is not equal to 1, and either uorder of vorder is greater than MAX_RATIONAL_EVAL_ORDER_NV.

The error INVALID_OPERATION is generated if EvalMapsNV is called where EVAL_VERTEX_ATTRIB0_NV is disabled.

New State

(add to table 6.22, page 212)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
EVAL_FRACTIONAL_TESSELLATION_NV	B	IsEnabled	False	fractional tess. enable	5.7	eval/enable
EVAL_VERTEX_ATTRIBi_NV	Bx16	IsEnabled	True if i=0, false otherwise	attrib eval enable	5.7	eval/enable
EVAL_2D_NV	R4x16x8*x8*	GetMapControlPointsNV	[0,0,0,1]	control points	5.7	-
EVAL_TRIANGULAR_2D_NV	R4x16x8*x8*	GetMapControlPoints	[0,0,0,1]	control points	5.7	-
MAP_TESSELLATION_NV	R4,R3	GetMapParameter*NV	all 1.0	level of tessellation	5.7	eval
MAP_ATTRIB_U_ORDER_NV	Z8*x16x2	GetMapAttribParameter*NV	1	map order in U direction	5.7	-
MAP_ATTRIB_V_ORDER_NV	Z8*x16x2	GetMapAttribParameter*NV	1	map order in V direction	5.7	-

New Implementation Dependent State

(add to table 6.24/6.25, page 214)

Get Value	Type	Get Command	Minimum Value	Description	Sec	Attribute
MAX_MAP_TESSELLATION_NV	Z+	GetIntegerv	256	maximum level of tessellation	5.7	-
MAX_RATIONAL_EVAL_ORDER_NV	Z+	GetIntegerv	4	maximum order of rational surfaces	5.7	-

Revision History

none yet

Name

    NV_fence

Name Strings

    GL_NV_fence

Notice

IP Status

    NVIDIA Proprietary.

Status

    Shipping as of June 8, 2000 (version 1.0)
    Shipping as of November, 2003 (version 1.1)

Version

    October 3, 2003 (version 1.1)
    $Id: //sw/main/docs/OpenGL/specs/GL_NV_fence.txt#13 $

Number

    222

Dependencies

    None

Overview

The goal of this extension is provide a finer granularity of synchronizing GL command completion than offered by standard OpenGL, which offers only two mechanisms for synchronization: Flush and Finish. Since Flush merely assures the user that the commands complete in a finite (though undetermined) amount of time, it is, thus, of only modest utility. Finish, on the other hand, stalls CPU execution until all pending GL commands have completed. This extension offers a middle ground - the ability to "finish" a subset of the command stream, and the ability to determine whether a given command has completed or not.

This extension introduces the concept of a "fence" to the OpenGL command stream. Once the fence is inserted into the command stream, it can be queried for a given condition - typically, its completion. Moreover, the application may also request a partial Finish -- that is, all commands prior to the fence will be forced to complete until control is returned to the calling process. These new mechanisms allow for synchronization between the host CPU and the GPU, which may be accessing the same resources (typically memory).

This extension is useful in conjunction with NV_vertex_array_range to determine when vertex information has been pulled from the vertex array range. Once a fence has been tested TRUE or finished, all vertex indices issued before the fence must have been pulled. This ensures that the vertex data memory corresponding to the issued vertex indices can be safely modified (assuming no other outstanding vertex indices are issued subsequent to the fence).

Issues

Do we need an IsFenceNV command?

RESOLUTION: Yes. Not sure who would use this, but it's in there. Semantics currently follow the texture object definition -- that is, calling IsFenceNV before SetFenceNV will return FALSE.

Are the fences sharable between multiple contexts?

RESOLUTION: No.

Potentially this could change with a subsequent extension.

What other conditions will be supported?

Only ALL_COMPLETED_NV will be supported initially. Future extensions may wish to implement additional fence conditions.

What is the relative performance of the calls?

Execution of a SetFenceNV is not free, but will not trigger a Flush or Finish.

Is the TestFenceNV call really necessary? How often would this be used compared to the FinishFenceNV call (which also flushes to ensure this happens in finite time)?

It is conceivable that a user may use TestFenceNV to decide which portion of memory should be used next without stalling the CPU. An example of this would be a scenario where a single AGP buffer is used for both static (unchanged for multiple frames) and dynamic (changed every frame) data. If the user has written dynamic data to all banks dedicated to dynamic data, and still has more dynamic objects to write, the user would first want to check if the first dynamic object has completed, before writing into the buffer. If the object has not completed, instead of stalling the CPU with a FinishFenceNV call, it would possibly be better to start overwriting static objects instead.

What should happen if TestFenceNV is called for a name before SetFenceNV is called?

We generate an INVALID_OPERATION error, and return TRUE. This follows the semantics for texture object names before they are bound, in that they acquire their state upon binding. We will arbitrarily return TRUE for consistency.

What should happen if FinishFenceNV is called for a name before SetFenceNV is called?

RESOLUTION: Generate an INVALID_OPERATION error because the fence id does not exist yet. SetFenceNV must be called to create a fence.

Do we need a mechanism to query which condition a given fence was set with?

RESOLUTION: Yes, use glGetFenceivNV with FENCE_CONDITION_NV.

Should we allow these commands to be compiled within display list? Which ones? How about within Begin/End pairs?

RESOLUTION: DeleteFencesNV, FinishFenceNV, GenFencesNV, TestFenceNV, and IsFenceNV are executed immediately while SetFenceNV is compiled. Do not allow any of these commands within Begin/End pairs.

Can fences be used as a form of performance monitoring?

Yes, with some caveats. By setting and testing or finishing fences, developers can measure the GPU latency for completing GL operations. For example, developers might do the following:

```
start = getCurrentTime();
updateTextures();
glSetFenceNV(TEXTURE_LOAD_FENCE, GL_ALL_COMPLETED_NV);
drawBackground();
glSetFenceNV(DRAW_BACKGROUND_FENCE, GL_ALL_COMPLETED_NV);
drawCharacters();
glSetFenceNV(DRAW_CHARACTERS_FENCE, GL_ALL_COMPLETED_NV);

glFinishFenceNV(TEXTURE_LOAD_FENCE);
textureLoadEnd = getCurrentTime();

glFinishFenceNV(DRAW_BACKGROUND_FENCE);
drawBackgroundEnd = getCurrentTime();

glFinishFenceNV(DRAW_CHARACTERS_FENCE);
drawCharactersEnd = getCurrentTime();

printf("texture load time = %d\n", textureLoadEnd - start);
printf("draw background time = %d\n", drawBackgroundEnd - textureLoadEnd);
printf("draw characters time = %d\n", drawCharacters - drawBackgroundEnd);
```

Note that there is a small amount of overhead associated with inserting each fence into the GL command stream. Each fence causes the GL command stream to momentarily idle (idling the entire GPU pipeline). The significance of this idling should be small if there are a small number of fences and large amount of intervening commands.

If the time between two fences is zero or very near zero, it probably means that a GPU-CPU synchronization such as a glFinish probably occurred. A glFinish is an explicit GPU-CPU synchronization, but sometimes implicit GPU-CPU synchronizations are performed by the driver.

What happens if you set the same fence object twice?

The second SetFenceNV clobbers whatever status the fence object previously had by forcing the object's status to GL_TRUE. The completion of the first SetFenceNV's fence command placed in the command stream is ignored (its completion does NOT update the fence object's status). The second SetFenceNV sets a new fence command in the GL command stream. This second fence command will update the fence object's status (assuming it is not ignored by a subsequent SetFenceNV to the same fence object).

What happens to a fence command that is still pending execution when its fence object is deleted?

The fence command completion is ignored.

New Procedures and Functions

```
    void GenFencesNV(sizei n, uint *fences);

    void DeleteFencesNV(sizei n, const uint *fences);

    void SetFenceNV(uint fence, enum condition);

    boolean TestFenceNV(uint fence);

    void FinishFenceNV(uint fence);

    boolean IsFenceNV(uint fence);

    void GetFenceivNV(uint fence, enum pname, int *params);
```

New Tokens

Accepted by the <condition> parameter of SetFenceNV:

```
        ALL_COMPLETED_NV                  0x84F2
```

Accepted by the <pname> parameter of GetFenceivNV:

```
        FENCE_STATUS_NV                   0x84F3
        FENCE_CONDITION_NV                0x84F4
```

Additions to Chapter 5 of the OpenGL 1.2.1 Specification (Special Functions)

Add to the end of Section 5.4 "Display Lists"

"DeleteFencesNV, FinishFenceNV, GenFencesNV, GetFenceivNV, TestFenceNV, and IsFenceNV are not complied into display lists but are executed immediately."

After the discussion of Flush and Finish (Section 5.5) add a description of the fence operations:

"5.X Fences

The command

```
   void SetFenceNV(uint fence, enum condition);
```

creates a fence object named <fence> if one does not already exist and sets a fence command within the GL command stream. If the named fence object already exists, a new fence command is set within the GL command stream (and any previous pending fence command corresponding to the fence object is ignored). Whether or not a new fence object is created, SetFenceNV assigns the named fence object a status of FALSE and a condition as set by the condition argument. The condition argument must be ALL_COMPLETED_NV. Once the fence's condition is satisfied within the command stream, the corresponding fence object's state is changed to TRUE. For a condition of ALL_COMPLETED_NV, this is completion of the fence command and all preceding commands. No other state is affected by execution of the fence command.

A fence's state can be queried by calling the command

```
  boolean TestFenceNV(uint fence);
```

The command

```
  void FinishFenceNV(uint fence);
```

forces all GL commands prior to the fence to satisfy the condition set within SetFenceNV, which, in this spec, is always completion. FinishFenceNV does not return until all effects from these commands on GL client and server state and the framebuffer are fully realized.

The fence must first be created before it can be used. The command

```
  void GenFencesNV(sizei n, uint *fences);
```

returns n previously unused fence names in fences. These names are marked as used, for the purposes of GenFencesNV only, but acquire boolean state only when they have been set.

Fences are deleted by calling

```
  void DeleteFencesNV(sizei n, const uint *fences);
```

fences contains n names of fences to be deleted. After a fence is deleted, it has no state, and its name is again unused. Unused names in fences are silently ignored.

If the fence passed to TestFenceNV or FinishFenceNV is not the name of a fence, the error INVALID_OPERATION is generated. In this case, TestFenceNV will return TRUE, for the sake of consistency.

State must be maintained to indicate which fence integers are currently used or set. In the initial state, no indices are in use. When a fence integer is set, the condition and status of the fence

    are also maintained. The status is a boolean. The condition is
    the value last set as the condition by SetFenceNV.

    Once the status of a fence has been finished (via FinishFenceNV)
    or tested and the returned status is TRUE (via either TestFenceNV
    or GetFenceivNV querying the FENCE_STATUS_NV), the status remains
    TRUE until the next SetFenceNV of the fence."

Additions to Chapter 6 of the OpenGL 1.2.1 Specification (State and State Requests)

    Insert new section after Section 6.1.10 "Minmax Query"

"6.1.11 Fence Query

    The command

```
      boolean IsFenceNV(uint fence);
```

    return TRUE if texture is the name of a fence. If fence is not the
    name of a fence, or if an error condition occurs, IsFenceNV returns
    FALSE. A name returned by GenFencesNV, but not yet set via SetFenceNV,
    is not the name of a fence.

    The command

```
      void GetFenceivNV(uint fence, enum pname, int *params)
```

    obtains the indicated fence state for the specified fence in the array
    params. pname must be either FENCE_STATUS_NV or FENCE_CONDITION_NV.
    The INVALID_OPERATION error is generated if the named fence does
    not exist."

Additions to the GLX Specification

    None

GLX Protocol

    Seven new GL commands are added.

    The following two rendering commands are sent to the sever as part
    of a glXRender request:

```
        SetFenceNV
            2           12              rendering command length
            2           4143            rendering command opcode
            4           CARD32          fence
            4           CARD32          condition
```

```
    The remaining five commands are non-rendering commands.  These
    commands are sent separately (i.e., not as part of a glXRender or
    glXRenderLarge request), using the glXVendorPrivateWithReply request:

        DeleteFencesNV
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           4+n             request length
            4           1276            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           INT32           n
            n*4         LISTofCARD32    fences

        GenFencesNV
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           4               request length
            4           1277            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           INT32           n
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           n               reply length
            24                          unused
            n*4         LISTofCARD322   fences

        IsFenceNV
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           4               request length
            4           1278            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           INT32           n
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           0               reply length
            4           BOOL32          return value
            20                          unused
        TestFenceNV
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           4               request length
            4           1279            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           INT32           fence
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           0               reply length
            4           BOOL32          return value
            20                          unused

        GetFenceivNV
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           5               request length
            4           1280            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           INT32           fence
            4           CARD32          pname
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           m               reply length, m=(n==1?0:n)
            4                           unused
            4           CARD32          n

            if (n=1) this follows:

            4           INT32           params
            12                          unused

            otherwise this follows:

            16                          unused
            n*4         LISTofINT32     params

        FinishFenceNV
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           4               request length
            4           1312            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           INT32           fence
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           0               reply length
            24                          unused
```

Errors

    INVALID_VALUE is generated if GenFencesNV parameter <n> is negative.

    INVALID_VALUE is generated if DeleteFencesNV parameter <n> is negative.

    INVALID_OPERATION is generated if the fence used in TestFenceNV or
    FinishFenceNV is not the name of a fence.

    INVALID_ENUM is generated if the condition used in SetFenceNV
    is not ALL_COMPLETED_NV.

    INVALID_OPERATION is generated if any of the commands defined in
    this extension is executed between the execution of Begin and the
    corresponding execution of End.

    INVALID_OPERATION is generated if the named fence in GetFenceivNV
    does not exist.

    INVALID_VALUE is generated if DeleteFencesNV or GenFencesNV are
    called where n is negative.

New State

Table 6.X. Fence Objects.

Get value	Type	Get command	Initial value	Description	Section	Attribute
FENCE_STATUS_NV	B	GetFenceivNV	determined by 1st SetFenceNV	Fence status	5.X	-
FENCE_CONDITION_NV	Z1	GetFenceivNV	determined by 1st SetFenceNV	Fence condition	5.X	-

New Implementation Dependent State

    None

GeForce Implementation Details

    This section describes implementation-defined limits for GeForce:

        SetFenceNV calls are not free. They should be used prudently,
        and a "good number" of commands should be sent between calls to
        SetFenceNV. Each fence insertion will cause the GPU's command
        processing to go momentarily idle. Testing or finishing a fence
        may require an one or more somewhat expensive uncached reads.

        Do not leave a fence untested or unfinished for an extremely large
        interval of intervening fences. If more than approximately 2
        billion (specifically 2^31-1) intervening fences are inserted into
        the GL command stream before a fence is tested or finished, said
        fence may indicate an incorrect status. Note that certain GL
        operations involving display lists, compiled vertex arrays, and
        textures may insert fences implicitly for internal driver use.

        In practice, this limitation is unlikely to be a practical
        limitation if fences are finished or tested within a few frames
        of their insertion into the GL command stream.

Revision History

November 13, 2000 - GLX enumerant values assigned

October 3, 2003 - Changed version to 1.1. glFinishFenceNV should not be compiled into display lists but rather executed immediately when called during display list construction. Version 1.0 allowed this though it should not have been allowed. Changed GLX protocol so that FinishFenceNV is a non-render request with a reply now. Thanks to Bob Beretta for noticing this issue.

Also fix a typo in the GLX protocol specification for IsFenceNV so the reply is 32 (not 33) bytes.

Name

    NV_float_buffer

Name Strings

    GL_NV_float_buffer
    WGL_NV_float_buffer

Notice

Status

    Implemented in CineFX (NV30) Emulation driver, August 2002.
    Shipping in Release 40 NVIDIA driver for CineFX hardware, January 2003.

Version

```
    Last Modified:      $Date: 2003/06/16 $
    NVIDIA Revision:    Revision: #16
```

Number

    281

Dependencies

    Written based on the wording of the OpenGL 1.3 specification and the
    WGL_ARB_pixel_format extension specification.

    The following extensions are required:
        * NV_fragment_program
        * NV_texture_rectangle
        * WGL_ARB_pixel_format
        * WGL_ARB_render_texture
        * WGL_NV_render_texture_rectangle

    EXT_paletted_texture trivially affects the definition of this extension.

    SGIX_depth_texture trivially affects the definition of this extension.

    NV_texture_shader trivially affects the definition of this extension.

    NV_half_float trivially affects the definition of this extension.

Overview

    This extension builds upon NV_fragment_program to provide a framebuffer
    and texture format that allows fragment programs to read and write
    unconstrained floating point data.

    In unextended OpenGL, most computations dealing with color or depth
    buffers are typically constrained to operate on values in the range [0,1].
    Computational results are also typically clamped to the range [0,1].

Color, texture, and depth buffers themselves also hold values mapped to the range [0,1].

The NV_fragment_program extension provides a general computational model that supports floating-point numbers constrained only by the precision of the underlying data types. The quantites computed by fragment programs do not necessarily correspond in number or in range to conventional attributes such as RGBA colors or depth values. Because of the range and precision constraints imposed by conventional fixed-point color buffers, it may be difficult (if not impossible) to use them to implement certain multi-pass algorithms.

To enhance the extended range and precision available through fragment programs, this extension provides floating-point RGBA color buffers that can be used instead of conventional fixed-point RGBA color buffers. A floating-point RGBA color buffer consists of one to four floating-point components stored in the 16- or 32-bit floating-point formats (fp16 or fp32) defined in the NV_half_float and NV_fragment_program extensions.

When a floating-point color buffer is used, the results of fragment programs, as written to the "x", "y", "z", and "w" components of the o[COLR] or o[COLH] output registers, are written directly to the color buffer without any clamping or modification. Certain per-fragment operations are bypassed when rendering to floating-point color buffers.

A floating-point color buffer can also be used as a texture map, either by reading back the contents and then using conventional TexImage calls, or by using the buffer directly via the ARB_render_texture extension.

This extension has many uses. Some possible uses include:

(1) Multi-pass algorithms with arbitrary intermediate results that don't have to be artifically forced into the range [0,1]. In addition, intermediate results can be written without having to worry about out-of-range values.

(2) Deferred shading algorithms where an expensive fragment program is executed only after depth testing is fully complete. Instead, a simple program is executed, which stores the parameters necessary to produce a final result. After the entire scene is rendered, a second pass is executed over the entire frame buffer to execute the complex fragment program using the results written to the floating-point color buffer in the first pass. This will save the cost of applying complex fragment programs to fragments that will not appear in the final image.

(3) Use floating-point texture maps to evaluate functions with arbitrary ranges. Arbitrary functions with a finite domain can be approximated using a texture map holding sample results and piecewise linear approximation.

There are several significant limitations on the use of floating-point color buffers. First, floating-point color buffers do not support frame buffer blending. Second, floating-point texture maps do not support mipmapping or any texture filtering other than NEAREST. Third, floating-point texture maps must be 2D, and must use the NV_texture_rectangle extension.

Issues

Should the extension create a separate non-RGBA pixel formats or simply extend existing RGBA formats?

RESOLVED: Extend existing RGBA formats. Since fragment programs generally build on RGBA semantics, it's cleaner to avoid creating a separate "XYZW" mode. There are several special semantics that need to be added: clear color state is now not clamped, and ReadPixels will clamp to [0,1] only if the source data comes from fixed-point color buffers.

Fragment programs can be written that store data completely unrelated to color into a floating-point "RGBA" buffer.

Can floating-point color buffers be displayed? If so, how?

RESOLVED: Not in this extension. Floating-point color buffers can be used only as pbuffers. Hardware necessary to display floating-point color buffers would be expensive and consume significant memory bandwidth.

Is it possible to encode more than four distinct values in a floating-point color buffer?

RESOLVED: Yes. The NV_fragment_program extension contains pack and unpack instructions (PK2H, PK2US, PK4B, PK4UB, PK4UBG, UP2H, UP2US, UP4B, UP4UB, UP4UBG) that allow fragment programs to encode multiple values into a single 32-bit component. In particular, it is possible to pack two half-precision floats, two normalized unsigned shorts, or four normalized signed or unsigned bytes into a single 32-bit component.

A program can use a pack instruction to pack multiple values into a single 32-bit component and then write the resulting component to a floating-point color buffer with 32-bit components. On a subsequent rendering pass, a program can read back the stored data (using texture mapping) and use the equivalent unpack instruction to restore the original values. The only data lost in this process comes from the loss of precision or clamping in the packing operation, where the original values are converted to data types with lower precision or a smaller data range.

What happens when rendering to an floating-point color buffer if fragment program mode is disabled? Or when fragment program mode is enabled, but no program is loaded?

RESOLVED: Fragment programs are required to use floating-point color buffers. An INVALID_OPERATION error is generated by any GL command that generates fragments if FRAGMENT_PROGRAM_NV is disabled. The same behavior already exists for conventional frame buffers if FRAGMENT_PROGRAM_NV is enabled but the bound fragment program is invalid.

Should alpha test be supported with floating-point color buffers?

RESOLVED: No. It is trivial to implement an alpha test in a fragment program using the KIL instruction, which requires no dedicated frame buffer logic.

Should blending be supported with floating-point color buffers?

RESOLVED: Not in this extension. While blending would clearly be useful, full-precision floating-point blenders are expensive. In addition, a computational model more general than traditional blending (with its 1-x operations and clamping) is desirable. The traditional OpenGL blending model would not be the most suitable computational model for future blend-enabled floating-point color buffers.

An alternative to conventional blending (operating at a coarser granularity) is to (1) render a pass into the color buffer, (2) bind the color buffer as a texture rectangle using this extension and ARB_render_texture, (3) perform texture lookups in a fragment program using the TEX instruction with f[WPOS].xy as a 2D texture coordinate, and (4) perform the necessary blending between the passes using the same fragment program.

Should we provide accumulation buffers for pixel formats with floating-point color buffers?

RESOLVED: No. Accumulation operations contents can be achieved using fragment programs to perform the accumulation, which requires no dedicated frame buffer logic.

Should fragment program color results be converted to match the format of the frame buffer, or should an error result? For example, what if we write to o[COLR] but have a 16-bit frame buffer?

RESOLVED: Conversions can be performed simply in hardware, so no error semantics are required. This mechanism also allows the same programs to be shared between contexts with different pixel formats.

Applications should be aware that if color components contain packed data, a data type mismatch may result in a floating-point data conversion that corrupts the packed data.

How should floating-point color buffers interact with multisampling? For normal color buffers, the multiple samples for each pixel are required to be filtered down to a single pixel in the color buffer. Similar filtering on floating-point color buffers does not necessarily make sense. Should there even be a normal color buffer in this case?

RESOLVED: The initial implementation of this extension does not provide floating-point color buffers that support multisampling.

Multisample fragment operations (e.g., SAMPLE_COVERAGE) are explicitly not supported by extension. This extension does not modify the portion of the spec where multiple samples are resolved to a single color value. So if floating-point color buffers were provided, the multiple samples are filtered down to a single result value, most likely by computing a per-component average value.

Conventional RGBA primitive antialiasing multiplies coverage by the alpha component of the fragment's color, with the assumption that alpha blending will be performed. How does antialiasing work with floating-point color buffers?

RESOLVED: It doesn't. The computed coverage is not accessible to fragment programs and is discarded. Note also that conventional antialiasing requires alpha blending, which does not work for floating-point color buffers.

What are the semantics for ReadPixels when using an floating-point color buffer?

RESOLVED: ReadPixels from a floating-point color buffer works like any other RGBA read, except that the final results are not clamped to the range [0,1]. This ensures that we can save and restore floating-point color buffers using ReadPixels/DrawPixels.

What are the semantics for Bitmap when using an floating-point color buffer?

RESOLVED: Bitmap generates fragments using the current raster attributes, which are then passed to fragment programs like any other fragments. Bitmaps will be drawn using the color of the current raster position, whose components are clamped to [0,1] when the raster position is sent.

What are the semantics for DrawPixels when using a floating-point color buffer? How about CopyPixels?

RESOLVED: DrawPixels generates fragments with the originally specified color values; components are not clamped to [0,1]. For fixed-point color buffers, DrawPixels will generate fragments with clamped color components.

CopyPixels is defined in the spec as a ReadPixels followed by a DrawPixels, and will operate similarly.

This mechanism allows applications to write floating-point data directly into a floating-point color buffer without any clamping. Since DrawPixels and CopyPixels generate fragments and fragment programs are required to render to floating-point color buffers, a fragment program is still required to load a floating-point color buffer using DrawPixels.

What are the semantics for Clear when using an floating-point color buffer?

RESOLVED: Clears work as normal, except that values outside the range [0,1] can be written to the color buffer. The core spec is modified so that clear color values are not clamped to [0,1]. Instead, for fixed-point color buffers, clear colors are clamped to [0,1] at clear time.

For compatibility with conventional OpenGL, queries of CLEAR_COLOR_VALUE will clamp components to [0,1]. A separate

FLOAT_CLEAR_COLOR_VALUE_NV query is added to query unclamped color clear values.

Why don't floating-point textures support filtering? What can be done to achieve texture filtering?

RESOLVED: Extended OpenGL texture filtering (including mipmapping and support for anisotropic filters) is very computationally expensive. Even simple linear filtering for floating-point textures with large components is expensive.

Linear filters can be implemented in fragment programs by doing multiple lookups into the same texture. Since fragment programs allow the use of arbitrary coordinates into arbitrary texture maps, this type of operation can be easily done.

A 1D linear filter can be implemented using an nx1 texture rectangle with the following (untested) fragment program, assuming the 1D coordinate is in f[TEX0].x:

```
ADDR H2.xy, f[TEX0].x, {0.0, 1.0};
FRCH H3.x, R1.x;             # compute the blend factor
TEX  H0, H2.x, TEX0, RECT;   # lookup 1st sample
TEX  H1, H2.y, TEX0, RECT;   # lookup 2nd sample
LRPH H0, H3.x, H1, H0;       # blend
```

A 2D linear filter can be implemented similarly, assuming the 2D coordinate is in f[TEX0].xy:

```
ADDH H2, f[TEX0].xyxy, {0.0, 0.0, 1.0, 1.0};
FRCH H3.xy, H2.xyxy;         # base weights
ADDH H3.zw, 1.0, -H3.xyxy;   # 1-base weights
MULH H3, H3.xzxz, H3.yyww;   # bilinear filter weights
TEX H1, R2.xyxy, TEX0, RECT; # lookup 1st sample
MULH H0, H1, H3.x;           # blend
TEX H1, R2.zyzy, TEX0, RECT; # lookup 2nd sample
MADH H0, H1, H3.y, H0;       # blend
TEX H0, R2.xwxw, TEX0, RECT; # lookup 3rd sample
MADH H0, H1, H3.z, H0;       # blend
TEX H1, R2.zwzw, TEX0, RECT; # lookup 4th sample
MADH H0, H1, H3.w, H0;       # blend
```

Fragment programs can be used to perform more-or-less arbitrary filtering using similar methods, and the DDX and DDY instructions can be used to refine the shape of the filter.

Why must the NV_texture_rectangle extension be used in order to use floating-point texture maps?

RESOLVED: On many graphics hardware platforms, texture maps are stored using a special memory encodings designed to optimize rendering performance. In current hardware, conventional texture maps usually top out at 32 bits per texel. The logic required to encode and decode 128-bit texels (and frame buffer pixels) optimally is substantially more complex.

What happens if you try to use an floating-point texture without a fragment program?

RESOLVED: No error is generated, but that texture is effectively disabled. This is similar to the behavior if an application tried to use a normal texture having an inconsistent set of mipmaps.

How does NV_float_buffer interact with the OpenGL 1.2 imaging subset?

RESOLVED: The imaging subset as specified should work properly with floating-point color buffers, but is not modified by this extension. There are imaging operations (e.g., color tables, histograms) that expect the components they operate on to be in the range [0,1], and this extension makes no attempt to extend such functionality.

How does NV_float_buffer interact with SGIS_generate_mipmap?

RESOLVED: Since this extension supports only texture rectangles (which have no mipmaps), this issue is moot.

In the general case, mipmaps should be generated using an appropriate downsample filter, where floating-point component values are averaged. Components should not be clamped during any such mipmap generation.

What is the deal with the names of the clear color query tokens?

RESOLVED: The "normal" OpenGL clear color (clamped to [0,1]) is queried using the token COLOR_CLEAR_VALUE. This extension provides a new query for unclamped values, using the token FLOAT_CLEAR_COLOR_VALUE_NV. Notice that "CLEAR" and "COLOR" are reversed due to a mistake made when the spec was first written. This spec lists the core query token, and originally had "CLEAR" and "COLOR" reversed there, too.

Then again, the core specification is inconsistent since the queried state is set by calling glClearColor(), with "Clear" before "Color".

What performance issues exist with this functionality?

See the "NV3x Implementation Issues" section of the specification.

How should the texture border color (values) be handled for float textures?

RESOLVED: Clamp the texture border color (values) to [0,1] when sampling a float texture's border. In core OpenGL 1.0, the texture border color components are clamped to the range [01,]. The NV_texture_shader extension added support for signed texture components. We decided to provide GL_TEXTURE_BORDER_VALUES as a way of specifying a version of the texture border color whose components were not clamped to [0,1] when set. This was to provide a way of specifying negative texture border components.

In practice, that has not proven particularly useful. No real applications are known to have specified negative texture border values components.

        Ideally, the unclamped GL_TEXTURE_BORDER_VALUES state could
        provide an unclamped (unmassaged) set of floating-point color
        components for the texture border color. This requires an
        additional 96 bits of state per texture unit to support this,
        and based on the experience with NV_texture_shader's support for
        texture border values outside the [0,1] range, it is simply not
        worth it.

        For compatibility with the NV_texture_shader extension, we
        provide language saying that floating-point textures clamp
        the components of the TEXTURE_BORDER_VALUES vector [0,1] when
        sampling the border color.

New Procedures and Functions

None.

New Tokens

Accepted by the <internalformat> parameter of TexImage2D and CopyTexImage2D:

```
        FLOAT_R_NV                                      0x8880
        FLOAT_RG_NV                                     0x8881
        FLOAT_RGB_NV                                    0x8882
        FLOAT_RGBA_NV                                   0x8883
        FLOAT_R16_NV                                    0x8884
        FLOAT_R32_NV                                    0x8885
        FLOAT_RG16_NV                                   0x8886
        FLOAT_RG32_NV                                   0x8887
        FLOAT_RGB16_NV                                  0x8888
        FLOAT_RGB32_NV                                  0x8889
        FLOAT_RGBA16_NV                                 0x888A
        FLOAT_RGBA32_NV                                 0x888B
```

Accepted by the <pname> parameter of GetTexLevelParameterfv and GetTexLevelParameteriv:

```
        TEXTURE_FLOAT_COMPONENTS_NV                     0x888C
```

Accepted by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

```
        FLOAT_CLEAR_COLOR_VALUE_NV                      0x888D
        FLOAT_RGBA_MODE_NV                              0x888E
```

Accepted in the <piAttributes> array of wglGetPixelFormatAttribivARB and wglGetPixelFormatAttribfvARB and in the <piAttribIList> and <pfAttribFList> arrays of wglChoosePixelFormatARB:

```
        WGL_FLOAT_COMPONENTS_NV                         0x20B0
        WGL_BIND_TO_TEXTURE_RECTANGLE_FLOAT_R_NV        0x20B1
        WGL_BIND_TO_TEXTURE_RECTANGLE_FLOAT_RG_NV       0x20B2
        WGL_BIND_TO_TEXTURE_RECTANGLE_FLOAT_RGB_NV      0x20B3
        WGL_BIND_TO_TEXTURE_RECTANGLE_FLOAT_RGBA_NV     0x20B4
```

Accepted in the <piAttribIList> array of wglCreatePbufferARB and returned in the <value> parameter of wglQueryPbufferARB when <iAttribute> is WGL_TEXTURE_FORMAT_ARB:

WGL_TEXTURE_FLOAT_R_NV	0x20B5
WGL_TEXTURE_FLOAT_RG_NV	0x20B6
WGL_TEXTURE_FLOAT_RGB_NV	0x20B7
WGL_TEXTURE_FLOAT_RGBA_NV	0x20B8

Additions to Chapter 2 of the OpenGL 1.3 Specification (OpenGL Operation)

None.

Additions to Chapter 3 of the OpenGL 1.3 Specification (Rasterization)

Modify Section 3.6.4, Rasterization of Pixel Rectangles (p. 91)

(modify first paragraph of "Final Conversion", p. 102) ... For RGBA components, the final conversion depends on the format of the color buffer. If the components of the color buffer are fixed-point, each element is clamped to [0,1] and converted to fixed-point according to the rules given in section 2.13.9 (Final Color Processing). If the components of the color buffer are floating-point, the elements are not modified.

Modify Section 3.8.1, Texture Image Specification (p. 116)

(modify last paragaph, p. 116) The selected groups are processed exactly as for DrawPixels stopping just before final conversion. For textures with fixed-point RGBA internal formats, each R, G, B, A component is clamped to [0,1].

(modify first paragraph, p. 117) Components are then selected from the resulting pixel groups to obtain a texture with the base internal format specified by (or derived from) <internalformat>. Table 3.15 summarizes the mapping of pixel group values to texture components, ...

(add to end of first paragraph, p. 117) Specifying a value of <format> incompatible with <internalformat> produces the error INVALID_OPERATION. A pixel format and texture internal format are compatible if the pixel format can generate a pixel group of the type listed in the "Pixel Group Type" column of Table 3.15 in the row corresponding to the base internal format.

(add between first and second paragraphs, p.117) Textures with a base internal format of FLOAT_R_NV, FLOAT_RG_NV, FLOAT_RGB_NV, and FLOAT_RGBA_NV are known as floating-point textures. Floating-point textures are only supported for the TEXTURE_RECTANGLE_NV target. Specifying an floating-point texture with any other target will produce an INVALID_OPERATION error.

(modify last paragraph, p. 117) The internal component resolution is the number of bits allocated to each component in a texture image. If internalformat is specified as a base internal format, the GL stores the resulting texture with internal component resolutions of its own choosing. If a sized internal format is specified, the memory allocation per texture component is assigned by the GL to match the allocations listed in Table 3.16 as closely as possible. ...

(modify Table 3.15, p. 118 -- Respecify this table with all extensions relevant to texture formats supported by NVIDIA. For this extension, add four base internal formats.)

Base Internal Format	Pixel Group Type	Component Values	Internal Components
ALPHA	RGBA	A	A
LUMINANCE	RGBA	R	L
LUMINANCE_ALPHA	RGBA	R,A	L,A
INTENSITY	RGBA	R	I
RGB	RGBA	R,G,B	R,G,B
RGBA	RGBA	R,G,B,A	R,G,B,A
* COLOR_INDEX	CI	CI	CI
* DEPTH_COMPONENT	DEPTH	DEPTH	DEPTH
* HILO_NV	HILO	HI,LO	HI,LO
* DSDT_NV	TEXOFF	DS,DT	DS,DT
* DSDT_MAG_NV	TEXOFF	DS,DT,MAG	DS,DT,MAG
* DSDT_MAG_INTENSITY_NV	TEXOFF or RGBA	DS,DT,MAG,VIB	DS,DT,MAG,I
FLOAT_R_NV	RGBA	R	R (float)
FLOAT_RG_NV	RGBA	R,G	R,G (float)
FLOAT_RGB_NV	RGBA	R,G,B	R,G,B (float)
FLOAT_RGBA_NV	RGBA	R,G,B,A	R,G,B,A (float)

Table 3.15: Conversion from pixel groups to internal texture components. "Pixel Group Type" defines the type of pixel group required for the specified internal format. All internal components are stored as unsigned-fixed point numbers, except for DS/DT (signed fixed-point numbers) and floating-point R,G,B,A (signed floating-point numbers). See Section 3.8.12 for a description of texture components R, G, B, A, L, and I. See NV_texture_shader spec (Section 3.8.13) for a description of texture components HI, LO, DS, DT, and MAG.

* - indicates formats found in other extension specs: COLOR_INDEX in EXT_paletted texture; DEPTH_COMPONENT in SGIX_depth_texture; and HILO_NV, DSDT_NV, DSDT_MAG_NV, DSDT_MAG_INTENSITY_NV in NV_texture_shader.

(modify Table 3.16, p. 119 -- Respecify this table with all extensions relevant to sized texture internal formats supported by NVIDIA. For this extension, add eight sized internal formats.)

	Sized Int. Format	Base Int. Format	Component Name / Type-Size			
	ALPHA4	ALPHA	A/U4			
	ALPHA8	ALPHA	A/U8			
	ALPHA12	ALPHA	A/U12			
	ALPHA16	ALPHA	A/U16			
	LUMINANCE4	LUMINANCE	L/U4			
	LUMINANCE8	LUMINANCE	L/U8			
	LUMINANCE12	LUMINANCE	L/U12			
	LUMINANCE16	LUMINANCE	L/U16			
	LUMINANCE4_ALPHA4	LUMINANCE_ALPHA	A/U4	L/U4		
	LUMINANCE6_ALPHA2	LUMINANCE_ALPHA	A/U2	L/U6		
	LUMINANCE8_ALPHA8	LUMINANCE_ALPHA	A/U8	L/U8		
	LUMINANCE12_ALPHA4	LUMINANCE_ALPHA	A/U4	L/U12		
	LUMINANCE12_ALPHA12	LUMINANCE_ALPHA	A/U12	L/U12		
	LUMINANCE16_ALPHA16	LUMINANCE_ALPHA	A/U16	L/U16		
	INTENSITY4	INTENSITY	I/U4			
	INTENSITY8	INTENSITY	I/U8			
	INTENSITY12	INTENSITY	I/U12			
	INTENSITY16	INTENSITY	I/U16			
	R3_G3_B2	RGB	R/U3	G/U3	B/U2	
	RGB4	RGB	R/U4	G/U4	B/U4	
	RGB5	RGB	R/U5	G/U5	B/U5	
	RGB8	RGB	R/U8	G/U8	B/U8	
	RGB10	RGB	R/U10	G/U10	B/10	
	RGB12	RGB	R/U12	G/U12	B/U12	
	RGB16	RGB	R/U16	G/U16	B/U16	
	RGBA2	RGBA	R/U2	G/U2	B/U2	A/U2
	RGBA4	RGBA	R/U4	G/U4	B/U4	A/U4
	RGB5_A1	RGBA	R/U5	G/U5	B/U5	A/U1
	RGBA8	RGBA	R/U8	G/U8	B/U8	A/U8
	RGB10_A2	RGBA	R/U10	G/U10	B/U10	A/U2
	RGBA12	RGBA	R/U12	G/U12	B/U12	A/U12
	RGBA16	RGBA	R/U16	G/U16	B/U16	A/U16
*	COLOR_INDEX1_EXT	COLOR_INDEX	CI/U1			
*	COLOR_INDEX2_EXT	COLOR_INDEX	CI/U2			
*	COLOR_INDEX4_EXT	COLOR_INDEX	CI/U4			
*	COLOR_INDEX8_EXT	COLOR_INDEX	CI/U8			
*	COLOR_INDEX16_EXT	COLOR_INDEX	CI/U16			
*	DEPTH_COMPONENT16_SGIX	DEPTH_COMPONENT	Z/U16			
*	DEPTH_COMPONENT24_SGIX	DEPTH_COMPONENT	Z/U24			
*	DEPTH_COMPONENT32_SGIX	DEPTH_COMPONENT	Z/U32			
*	HILO16_NV	HILO	HI/U16	LO/U16		
*	SIGNED_HILO16_NV	HILO	HI/S16	LO/S16		
*	SIGNED_RGBA8_NV	RGBA	R/S8	G/S8	B/S8	A/S8
*	SIGNED_RGB8_ UNSIGNED_ALPHA8_NV	RGBA	R/S8	G/S8	B/S8	A/U8
*	SIGNED_RGB8_NV	RGB	R/S8	G/S8	B/S8	
*	SIGNED_LUMINANCE8_NV	LUMINANCE	L/S8			
*	SIGNED_LUMINANCE8_ ALPHA8_NV	LUMINANCE_ALPHA	L/S8	A/S8		
*	SIGNED_ALPHA8_NV	ALPHA	A/S8			
*	SIGNED_INTENSITY8_NV	INTENSITY	I/S8			
*	DSDT8_NV	DSDT_NV	DS/S8	DT/S8		
*	DSDT8_MAG8_NV	DSDT_MAG_NV	DS/S8	DT/S8	MAG/U8	
*	DSDT8_MAG8_ INTENSITY8_NV	DSDT_MAG_ INTENSITY_NV	DS/S8	DT/S8	MAG/U8	I/U8
	FLOAT_R16_NV	FLOAT_R_NV	R/F16			
	FLOAT_R32_NV	FLOAT_R_NV	R/F32			
	FLOAT_RG16_NV	FLOAT_RG_NV	R/F16	G/F16		
	FLOAT_RG32_NV	FLOAT_RG_NV	R/F32	G/F32		
	FLOAT_RGB16_NV	FLOAT_RGB_NV	R/F16	G/F16	B/F16	
	FLOAT_RGB32_NV	FLOAT_RGB_NV	R/F32	G/F32	B/F32	
	FLOAT_RGBA16_NV	FLOAT_RGBA_NV	R/F16	G/F16	B/F16	A/F16
	FLOAT_RGBA32_NV	FLOAT_RGBA_NV	R/F32	G/F32	B/F32	A/F32

Table 3.16: Sized Internal Formats. Describes the correspondence of sized internal formats to base internal formats, and desired component resolutions. Component resolution descriptions are of the form "<NAME>/<TYPE><SIZE>", where NAME specifies the component name in Table 3.15, TYPE is "U" for unsigned fixed-point, "S" for signed fixed-point, and "F" for unsigned floating-point. <SIZE> is the number of requested bits per component.

* - indicates formats found in other extension specs: COLOR_INDEX in EXT_paletted texture; DEPTH_COMPONENT in SGIX_depth_texture; and HILO_NV, DSDT_NV, DSDT_MAG_NV, DSDT_MAG_INTENSITY_NV in NV_texture_shader.

Modify Section 3.8,7, Minification (p. 141)

Change the last paragraph (as modified by the NV_texture_shader extension) to read (only the last sentence changes from the NV_texture_shader version):

"If any of the selected tauijk, tauij, or taui in the above equations refer to a border texel with i < -bs, j < bs, k < -bs, i >= ws-bs, j >= hs-bs, or k >= ds-bs, then the border values given by the current setting of TEXTURE_BORDER_VALUES is used instead of the unspecified value or values. If the texture contains color components, the components of the TEXTURE_BORDER_VALUES vector are interpreted as an RGBA color to match the texture's internal format in a manner consistent with table 3.15. If the texture contains HILO components, the first and second components of the TEXTURE_BORDER_VALUES vector are interpreted as the hi and lo components respectively. If the texture contains texture offset group components, the first, second, third, and fourth components of the TEXTURE_BORDER_VALUES vector are interpreted as ds, dt, mag, and vib components respectively. Additionally, the texture border values are clamped appropriately depending on the signedness of each particular component. Unsigned components and components of floating-point textures are clamped to [0,1]; signed components (not including floating-point textures) are clamped to [-1,1]."

(Add after the last paragraph in the section) Floating-point textures (those with a base internal format of FLOAT_R_NV, FLOAT_RG_NV, FLOAT_RGB_NV, or FLOAT_RGBA_NV) do not support texture filters other than NEAREST. For such textures, NEAREST filtering is applied regardless of the setting of TEXTURE_MIN_FILTER.

Modify Section 3.8.8, Magnification (p. 141)

(Add after the last paragraph in the section) Floating-point textures (those with a base internal format of FLOAT_R_NV, FLOAT_RG_NV, FLOAT_RGB_NV, or FLOAT_RGBA_NV) do not support texture filters other than NEAREST. For such textures, NEAREST filtering is applied regardless of the setting of TEXTURE_MAG_FILTER.

Modify Section 3.8.13, Texture Environments and Texture Functions (p. 147)

(Add paragraph after discussion of all the values used in the miscellaneous tables in this section.) If the base internal format is HILO_NV, DSDT_NV, DSDT_MAG_NV, DSDT_MAG_INTENSITY_NV, FLOAT_R_NV, FLOAT_RG_NV, FLOAT_RGB_NV, or FLOAT_RGBA_NV, the texture lookup results are not supported using conventional OpenGL texture functions. In this case, the corresponding texture function is NONE (Cv = Cf, Av = Af), and it is as though texture mapping were disabled for that texture unit.

Modify Section 3.11, Antialiasing Application (p. 155)

Finally, if antialiasing is enabled for the primitive from which a rasterized fragment was produced, then the computed coverage value may be applied to the fragment. In RGBA mode with fixed-point frame buffers, the value is multiplied by the fragment's alpha (A) value to yield a final alpha value. In RGBA mode with floating-point frame buffers, the coverage value is simply discarded. In color index mode, the value is used to set the low order bits of the color index value as described in section 3.2.

Additions to Chapter 4 of the OpenGL 1.3 Specification (Per-Fragment Operations and the Frame Buffer)

Modify Chapter 4 Introduction (p. 156)

(replace next-to-last paragraph)

The GL provides three types of color buffers: color index, fixed-point RGBA, or floating-point RGBA. Color index buffers consist of unsigned integer color indices. Fixed-point RGBA buffers consist of R, G, B, and optionally, A unsigned integer values. Floating-point RGBA buffers consist of R, and optionally, G, B, and A floating-point component values, corresponding to the X, Y, Z, and W outputs, respectively, of a fragment program. The number of bitplanes in each of the color buffers, the depth buffer, ...

Modify Section 4.1.3, Multisample Fragment Operations (p. 158)

This step applies only for fixed-point RGBA color buffers. Otherwise, proceed to the next step. ...

Modify Section 4.1.4, Alpha Test (p. 159)

This step applies only for fixed-point RGBA color buffers. Otherwise, proceed to the next step. ...

Modify Section 4.1.7, Blending (p. 161)

(modify second paragraph)

This blending is dependent on the incoming fragment's alpha value and that of the corresponding currently stored pixel. Blending applies only for fixed-point RGBA color buffers; otherwise, it is bypassed. ...

Modify Section 4.1.8, Dithering (p. 165)

Dithering selects between two color values or indices. Dithering does not apply to floating-point RGBA color buffers. ...

Modify Section 4.1.9, Logical Operation (p. 165)

Finally, a logical operation is applied between the incoming fragment's color or index values and the color or index values stored at the corresponding location in the frame buffer. Logical operations do not apply to floating-point color buffers. ...

Modify Section 4.2.3, Clearing the Buffers (p. 171)

...

```
    void ClearColor(float r, float g, float b, float a);
```

sets the clear value for RGBA color buffers. When a fixed-point color buffer is cleared, the effective clear color is derived by clamping each component to [0,1] and converting to fixed-point according to the rules in section 2.13.9. When a floating-point color buffer is cleared, the components of the clear value are used directly without being clamped.

Modify Section 4.2.4, The Accumulation Buffer (p. 172)

(modify last paragraph) ... If there is no accumulation buffer, or if color buffer is not fixed-point RGBA, Accum generates the error INVALID_OPERATION.

Modify Section 4.3.2, Reading Pixels

(modify "Conversion of RGBA Values", p. 176) This step applies only if the GL is in RGBA mode, and then only if format is neither STENCIL INDEX nor DEPTH COMPONENT. The R, G, B, and A values form a group of elements. If the color buffer has fixed-point format, each element is taken to be a fixed-point value in [0,1] with m bits, where m is the number of bits in the corresponding color component of the selected buffer (see section 2.13.9).

(add to end of "Final Conversion", p. 177) ... For an RGBA color, components are clamped depending on the data type of the buffer being read. For fixed-point buffers, each component is clamped to [0.1]. For floating-point buffers, if <type> is not FLOAT or HALF_FLOAT_NV, each component is clamped to [0,1] if <type> is unsigned or [-1,1] if <type> is signed and then converted according to Table 4.7.

Additions to Chapter 5 of the OpenGL 1.3 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 1.3 Specification (State and State Requests)

Modify Section 6.1.4, Texture Queries (p. 200)

Modify Table 6.1 (add new rows, corresponding to new internal formats, p. 202)

Base Internal Format	R	G	B	A
FLOAT_R_NV	R	0	0	1
FLOAT_RG_NV	R	G	0	1
FLOAT_RGB_NV	R	G	B	1
FLOAT_RGBA_NV	R	G	B	A

Additions to Appendix A of the OpenGL 1.3 Specification (Invariance)

None.

Additions to the WGL Specification

First, close your eyes and pretend that a WGL specification actually existed. Maybe if we all concentrate hard enough, one will magically appear.

```
    Modify/add to the description of <piAttributes> in
    wglGetPixelFormatAttribivARB and <pfAttributes> in
    wglGetPixelFormatAttribfvARB:

      WGL_FLOAT_COMPONENTS_NV
        True if the R, G, B, and A components of each color buffer are
        represented as (unclamped) floating-point numbers.

      WGL_BIND_TO_TEXTURE_RECTANGLE_FLOAT_R_NV
      WGL_BIND_TO_TEXTURE_RECTANGLE_FLOAT_RG_NV
      WGL_BIND_TO_TEXTURE_RECTANGLE_FLOAT_RGB_NV
      WGL_BIND_TO_TEXTURE_RECTANGLE_FLOAT_RGBA_NV
        True if the pixel format describes a floating-point color that can be
        bound to a texture rectangle with internal formats of FLOAT_R_NV,
        FLOAT_RG_NV, FLOAT_RGB_NV, or FLOAT_RGBA_NV, respectively.  Currently
        only pbuffers can be bound as textures so this attribute will only be
        TRUE if WGL_DRAW_TO_PBUFFER is also TRUE.  Additionally,
        floating-point color buffers can not be bound to texture targets other
        than TEXTURE_RECTANGLE_NV.

    Add new table entries for pixel format attribute matching in
    wglChoosePixelFormatARB.

        Attribute                    Type        Match Criteria
        -------------------------    -------     --------------
        WGL_FLOAT_COMPONENTS_NV      boolean     exact
        WGL_BIND_TO_TEXTURE_         boolean     exact
          RECTANGLE_FLOAT_R_NV
        WGL_BIND_TO_TEXTURE_         boolean     exact
          RECTANGLE_FLOAT_RG_NV
        WGL_BIND_TO_TEXTURE_         boolean     exact
          RECTANGLE_FLOAT_RGB_NV
        WGL_BIND_TO_TEXTURE_         boolean     exact
          RECTANGLE_FLOAT_RGBA_NV

    (In the wglCreatePbufferARB section, modify the attribute list)

      WGL_TEXTURE_FORMAT_ARB

        This attribute indicates the base internal format of the texture that
        will be created when a color buffer of a pbuffer is bound to a texture
        map.  It can be set to WGL_TEXTURE_RGB_ARB (indicating an internal
        format of RGB), WGL_TEXTURE_RGBA_ARB (indicating a base internal
        format of RGBA), WGL_TEXTURE_FLOAT_R_NV (indicating a base internal
        format of FLOAT_R_NV), WGL_TEXTURE_FLOAT_RG_NV (indicating a base
        internal format of FLOAT_RG_NV), WGL_TEXTURE_FLOAT_RGB_NV (indicating
        a base internal format of FLOAT_RGB_NV), WGL_TEXTURE_FLOAT_RGBA_NV
        (indicating a base internal format of FLOAT_RGBA_NV), or
        WGL_NO_TEXTURE_ARB. The default value is WGL_NO_TEXTURE_ARB.


    (In the wglCreatePbufferARB section, modify the discussion of what happens
    to the depth/stencil/accum buffers when switching between mipmap levels or
    cube map faces.)

    For pbuffers with a texture format of WGL_TEXTURE_RGB_ARB,
    WGL_TEXTURE_RGBA_ARB, WGL_TEXTURE_FLOAT_R_NV, WGL_TEXTURE_FLOAT_RG_NV,
    WGL_TEXTURE_FLOAT_RGB_NV, or WGL_TEXTURE_FLOAT_RGBA_NV, there will be a
    separate set of color buffers for each mipmap level and cube map face in
    the pbuffer.  Otherwise, the WGL implementation is free to share a single
    set of color, auxillary, and accumulation buffers between levels or faces.


    (In the wglCreatePbufferARB section, modify the error list)

        ERROR_INVALID_DATA      WGL_TEXTURE_FORMAT_ARB is
                                WGL_TEXTURE_FLOAT_R_NV,
                                WGL_TEXTURE_FLOAT_RG_NV,
                                WGL_TEXTURE_FLOAT_RGB_NV, or
                                WGL_TEXTURE_FLOAT_RGBA_NV, and
                                WGL_TEXTURE_TARGET_ARB is not
                                WGL_TEXTURE_RECTANGLE_NV.

        ERROR_INVALID_DATA      WGL_TEXTURE_FORMAT_ARB is
                                WGL_TEXTURE_FLOAT_R_NV,
                                WGL_TEXTURE_TARGET_ARB is
                                WGL_TEXTURE_RECTANGLE_NV, and the
                                WGL_BIND_TO_TEXTURE_RECTANGLE_FLOAT_R_NV
                                attribute is not set in the pixel format.

        ERROR_INVALID_DATA      WGL_TEXTURE_FORMAT_ARB is
                                WGL_TEXTURE_FLOAT_RG_NV,
                                WGL_TEXTURE_TARGET_ARB is
                                WGL_TEXTURE_RECTANGLE_NV, and the
                                WGL_BIND_TO_TEXTURE_RECTANGLE_FLOAT_RG_NV
                                attribute is not set in the pixel format.

        ERROR_INVALID_DATA      WGL_TEXTURE_FORMAT_ARB is
                                WGL_TEXTURE_FLOAT_RGB_NV,
                                WGL_TEXTURE_TARGET_ARB is
                                WGL_TEXTURE_RECTANGLE_NV, and the
                                WGL_BIND_TO_TEXTURE_RECTANGLE_FLOAT_RGB_NV
                                attribute is not set in the pixel format.

        ERROR_INVALID_DATA      WGL_TEXTURE_FORMAT_ARB is
                                WGL_TEXTURE_FLOAT_RGBA_NV,
                                WGL_TEXTURE_TARGET_ARB is
                                WGL_TEXTURE_RECTANGLE_NV, and the
                                WGL_BIND_TO_TEXTURE_RECTANGLE_FLOAT_RGBA_NV
                                attribute is not set in the pixel format.

    Modify wglBindTexImageARB:

    ...

        The pbuffer attribute WGL_TEXTURE_FORMAT_ARB determines the base
        internal format of the texture. The format-specific component sizes
        are also determined by pbuffer attributes as shown in the table below.
        The component sizes are dependent on the format of the texture.

        Component           Size                 Format
        ---------    ------------------------    ----------------------------
            R        WGL_RED_BITS_ARB            RGB, RGBA, FLOAT_R, FLOAT_RG,
                                                 FLOAT_RGB, FLOAT_RGBA
```

G	WGL_GREEN_BITS_ARB	RGB, RGBA, FLOAT_R, FLOAT_RG, FLOAT_RGB, FLOAT_RGBA
B	WGL_BLUE_BITS_ARB	RGB, RGBA, FLOAT_R, FLOAT_RG, FLOAT_RGB, FLOAT_RGBA
A	WGL_ALPHA_BITS_ARB	RGB, RGBA, FLOAT_R, FLOAT_RG, FLOAT_RGB, FLOAT_RGBA

Additions to the AGL/GLX Specification

None.

Dependencies on EXT_paletted_texture, SGIX_depth_texture, and NV_texture_shader

If any of these extensions are not supported, the rows in Tables 3.15 and 3.16 corresponding to texture formats defined by the unsupported extension should be removed.

If NV_texture_shader is not supported, ignore the amended paragraph from the NV_texture_shader specificiaton describing TEXTURE_BORDER_VALUES clamping in favor of the original OpenGL specification language.

Dependencies on NV_half_float

If GL_NV_half_float is not supported, all references to HALF_FLOAT_NV should be deleted.

GLX Protocol

None.

Errors

INVALID_OPERATION is generated by Begin, DrawPixels, Bitmap, CopyPixels, or a command that performs an explicit Begin if the color buffer has a floating-point RGBA format and FRAGMENT_PROGRAM_NV is disabled.

INVALID_OPERATION is generated by TexImage3D, TexImage2D, TexImage1D, TexSubImage3D, TexSubImage2D, or TexSubImage1D if the pixel group type corresponding to <format> is not compatible with the base internal format of the texture.

INVALID_OPERATION is generated by TexImage3D, TexImage1D, or CopyTexImage1D if the base internal format corresponding to <internalformat> is FLOAT_R_NV, FLOAT_RG_NV, FLOAT_RGB_NV, or FLOAT_RGBA_NV.

INVALID_OPERATION is generated by TexImage2D or CopyTexImage2D if the base internal format corresponding to <internalformat> is FLOAT_R_NV, FLOAT_RG_NV, FLOAT_RGB_NV, or FLOAT_RGBA_NV and <target> is not TEXTURE_RECTANGLE_NV.

INVALID_OPERATION is generated by Accum if the color buffer has a color index or floating-point RGBA format.

```
    ERROR_INVALID_DATA is generated by wglCreatePbufferARB if
    WGL_TEXTURE_FORMAT_ARB is WGL_TEXTURE_FLOAT_R_NV, WGL_TEXTURE_FLOAT_RG_NV,
    WGL_TEXTURE_FLOAT_RGB_NV, or WGL_TEXTURE_FLOAT_RGBA_NV, and
    WGL_TEXTURE_TARGET_ARB is not WGL_TEXTURE_RECTANGLE_NV.

    ERROR_INVALID_DATA is generated by wglCreatePbufferARB if
    WGL_TEXTURE_FORMAT_ARB is WGL_TEXTURE_FLOAT_R_NV, WGL_TEXTURE_TARGET_ARB
    is WGL_TEXTURE_RECTANGLE_NV, and the
    WGL_BIND_TO_TEXTURE_RECTANGLE_FLOAT_R_NV attribute is not set in the pixel
    format.

    ERROR_INVALID_DATA is generated by wglCreatePbufferARB if
    WGL_TEXTURE_FORMAT_ARB is WGL_TEXTURE_FLOAT_RG_NV, WGL_TEXTURE_TARGET_ARB
    is WGL_TEXTURE_RECTANGLE_NV, and the
    WGL_BIND_TO_TEXTURE_RECTANGLE_FLOAT_RG_NV attribute is not set in the
    pixel format.

    ERROR_INVALID_DATA is generated by wglCreatePbufferARB if
    WGL_TEXTURE_FORMAT_ARB is WGL_TEXTURE_FLOAT_RGB_NV, WGL_TEXTURE_TARGET_ARB
    is WGL_TEXTURE_RECTANGLE_NV, and the
    WGL_BIND_TO_TEXTURE_RECTANGLE_FLOAT_RGB_NV attribute is not set in the
    pixel format.

    ERROR_INVALID_DATA is generated by wglCreatePbufferARB if
    WGL_TEXTURE_FORMAT_ARB is WGL_TEXTURE_FLOAT_RGBA_NV,
    WGL_TEXTURE_TARGET_ARB is WGL_TEXTURE_RECTANGLE_NV, and the
    WGL_BIND_TO_TEXTURE_RECTANGLE_FLOAT_RGBA_NV attribute is not set in the
    pixel format.
```

New State

(Modify Table 6.15, Texture Objects (cont.), p. 223)

Get Value	Type	Get Command	Init. Value	Description	Sec.	Attribute
TEXTURE_FLOAT_COMPONENTS_NV	n x B	GetTexLevel-	0	True if texture holds unclamped floating-point values	3.8	-

(Modify Table 6.19, Framebuffer Control, p. 227)

Get Value	Type	Get Command	Init. Value	Description	Sec.	Attribute
COLOR_CLEAR_VALUE	C	GetFloatv	0,0,0,0	Color buffer clear value (RGBA mode), each value clamped to [0,1].	4.2.3	color-buffer
FLOAT_CLEAR_COLOR_VALUE_NV	4xR	GetFloatv	0,0,0,0	Color buffer clear value (RGBA mode), each value unclamped.	4.2.3	color-buffer

New Implementation Dependent State

(Modify Table 6.28, Implementation Dependent Values, p. 236)

Get Value	Type	Get Command	Init. Value	Description	Sec.	Attribute
FLOAT_RGBA_MODE_NV	B	GetBooleanv	-	True if color buffers store floating-point data	4	-

NV3x Implementation Details

NV3x GPUs (GeForce FX, etc.) support hardware acceleration for float textures with two or more components only when the repeat mode state (S and T) is GL_CLAMP_TO_EDGE. If you use either the GL_CLAMP or GL_CLAMP_TO_BORDER repeat modes with a float texture with two or more components, the software rasterizer is used.

However, if you use a single-component float texture (GL_FLOAT_R_NV, etc.), all clamping repeat modes (GL_CLAMP, GL_CLAMP_TO_EDGE, and GL_CLAMP_TO_BORDER) are available with full hardware acceleration.

The two-, three-, and four-component texture formats all use the same amount of texture memory storage (128 bits per texel for the GL_FLOAT_x32 formats, and 64 bits per texel for the GL_FLOAT_x16 formats). Future GPUs will likely store two and three component float textures more efficiently.

The GL_FLOAT_R32_NV and GL_FLOAT_R16_NV texture formats each use 32 bits per texel. Future GPUs will likely store GL_FLOAT_R16_NV more efficiently.

NVIDIA treats the unsized internal formats GL_FLOAT_R_NV, GL_FLOAT_RGBA_NV, etc. the same as GL_FLOAT_R32_NV, GL_FLOAT_RGBA32_NV, etc.

Revision History

Rev.	Date	Author	Changes
16	06/16/03	pbrown	Corrected the usage of WGL_TEXTURE_FLOAT_R_NV and related enums in the list of enumerants.
15	01/23/03	mjk	Document texture border color (values) behavior for float textures. See issue.
14	01/20/03	mjk	Added NV3x Implementation Details section.

Name

    NV_fog_distance

Name Strings

    GL_NV_fog_distance

Notice

IP Status

NVIDIA Proprietary.

Status

Shipping (version 1.0)

Version

NVIDIA Date: January 18, 2001
$Id: //sw/main/docs/OpenGL/specs/GL_NV_fog_distance.txt#14 $

Number

192

Dependencies

Written based on the wording of the OpenGL 1.2 specification.

Overview

Ideally, the fog distance (used to compute the fog factor as described in Section 3.10) should be computed as the per-fragment Euclidean distance to the fragment center from the eye. In practice, implementations "may choose to approximate the eye-coordinate distance from the eye to each fragment center by abs(ze). Further, [the fog factor] f need not be computed at each fragment, but may be computed at each vertex and interpolated as other data are."

This extension provides the application specific control over how OpenGL computes the distance used in computing the fog factor.

The extension supports three fog distance modes: "eye plane absolute", where the fog distance is the absolute planar distance from the eye plane (i.e., OpenGL's standard implementation allowance as cited above); "eye plane", where the fog distance is the signed planar distance from the eye plane; and "eye radial", where the fog distance is computed as a Euclidean distance. In the case of the eye radial fog distance mode, the distance may be computed per-vertex and then interpolated per-fragment.

The intent of this extension is to provide applications with better control over the tradeoff between performance and fog quality. The "eye planar" modes (signed or absolute) are straightforward to implement with good performance, but scenes are consistently under-fogged at the edges of the field of view. The "eye radial" mode can provide for more accurate fog at the edges of the field of view, but this assumes that either the eye radial fog distance is computed per-fragment, or if the fog distance is computed per-vertex and then interpolated per-fragment, then the scene must be sufficiently tessellated.

Issues

What should the default state be?

IMPLEMENTATION DEPENDENT.

The EYE_PLANE_ABSOLUTE_NV mode is the most consistent with the way most current OpenGL implementations are implemented without this extension, but because this extension provides specific control over a capability that core OpenGL is intentionally lax about, the default fog distance mode is left implementation dependent. We would not want a future OpenGL implementation that supports fast EYE_RADIAL_NV fog distance to be stuck using something less.

Advice: If an implementation can provide fast per-pixel EYE_RADIAL_NV support, then EYE_RADIAL_NV is the ideal default, but if not, then EYE_PLANE_ABSOLUTE_NV is the most reasonable default mode.

How does this extension interact with the EXT_fog_coord extension?

If FOG_COORDINATE_SOURCE_EXT is set to FOG_COORDINATE_EXT, then the fog distance mode is ignored. However, the fog distance mode is used when the FOG_COORDINATE_SOURCE_EXT is set to FRAGMENT_DEPTH_EXT. Essentially, when the EXT_fog_coord functionality is enabled, the fog distance is supplied by the user-supplied fog-coordinate so no automatic fog distance computation is performed.

New Procedures and Functions

None

New Tokens

Accepted by the <pname> parameters of Fogf, Fogi, Fogfv, Fogiv, GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

```
    FOG_DISTANCE_MODE_NV              0x855A
```

When the <pname> parameter of Fogf, Fogi, Foggv, and Fogiv, is FOG_DISTANCE_MODE_NV, then the value of <param> or the value pointed to by <params> may be:

```
    EYE_RADIAL_NV                     0x855B
    EYE_PLANE
    EYE_PLANE_ABSOLUTE_NV             0x855C
```

Additions to Chapter 2 of the 1.2 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.2 Specification (Rasterization)

-- Section 3.10 "Fog"

Add to the end of the 3rd paragraph:

"If pname is FOG_DISTANCE_MODE_NV, then param must be, or params must point to an integer that is one of the symbolic constants EYE_PLANE_ABSOLUTE_NV, EYE_PLANE, or EYE_RADIAL_NV and this symbolic constant determines how the fog distance should be computed."

Replace the 4th paragraph beginning "An implementation may choose to approximate ..." with:

"When the fog distance mode is EYE_PLANE_ABSOLUTE_NV, the fog distance z is approximated by abs(ze) [where ze is the Z component of the fragment's eye position]. When the fog distance mode is EYE_PLANE, the fog distance z is approximated by ze. When the fog distance mode is EYE_RADIAL_NV, the fog distance z is computed as the Euclidean distance from the center of the fragment in eye coordinates to the eye position. Specifically:

```
  z  =  sqrt( xe*xe + ye*ye + ze*ze );
```

In the EYE_RADIAL_NV fog distance mode, the Euclidean distance is permitted to be computed per-vertex, and then interpolated per-fragment."

Change the last paragraph to read:

"The state required for fog consists of a three valued integer to select the fog equation, a three valued integer to select the fog distance mode, three floating-point values d, e, and s, and RGBA fog color and a fog color index, and a single bit to indicate whether or not fog is enabled. In the initial state, fog is disabled, FOG_MODE is EXP, FOG_DISTANCE_NV is implementation defined, d = 1.0, e = 1.0, and s = 0.0; Cf = (0,0,0,0) and if = 0."

Additions to Chapter 4 of the 1.2 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.2 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

None

Additions to the GLX Specification

None

Errors

INVALID_ENUM is generated when Fog is called with a <pname> of FOG_DISTANCE_MODE_NV and the value of <param> or what is pointed to by <params> is not one of EYE_PLANE_ABSOLUTE_NV, EYE_PLANE, or EYE_RADIAL_NV.

New State

(table 6.8, p198) add the entry:

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
FOG_DISTANCE_MODE_NV	Z3	GetIntegerv	implementation dependent	Determines how fog distance is computed	3.10	fog

New Implementation State

None

Name

    NV_fragment_program

Name Strings

    GL_NV_fragment_program

Notice

    Copyright NVIDIA Corporation, 2001-2002.

IP Status

    NVIDIA Proprietary.

Status

    Implemented in CineFX (NV30) Emulation driver, August 2002.
    Shipping in Release 40 NVIDIA driver for CineFX hardware, January 2003.

Version

```
    Last Modified Date:  $Date: 2003/05/12 $
    NVIDIA Revision:     70
```

Number

    282

Dependencies

    Written based on the wording of the OpenGL 1.2.1 specification and
    requires OpenGL 1.2.1.

    Requires support for the ARB_multitexture extension with at least
    two texture units.

    NV_vertex_program affects the definition of this extension. The only
    dependency is that both extensions use the same mechanisms for defining
    and binding programs.

    NV_texture_shader trivially affects the definition of this extension.

    NV_texture_rectangle trivially affects the definition of this extension.

    ARB_texture_cube_map trivially affects the definition of this extension.

    EXT_fog_coord trivially affects the definition of this extension.

    NV_depth_clamp affects the definition of this extension.

    ARB_depth_texture and SGIX_depth_texture affect the definition of this
    extension.

NV_float_buffer affects the definition of this extension.

ARB_vertex_program affects the definition of this extension.

ARB_fragment_program affects the definition of this extension.

Overview

OpenGL mandates a certain set of configurable per-fragment computations defining texture lookup, texture environment, color sum, and fog operations. Each of these areas provide a useful but limited set of fixed operations. For example, unextended OpenGL 1.2.1 provides only four texture environment modes, color sum, and three fog modes. Many OpenGL extensions have either improved existing functionality or introduced new configurable fragment operations. While these extensions have enabled new and interesting rendering effects, the set of effects is limited by the set of special modes introduced by the extension. This lack of flexibility is in contrast to the high-level of programmability of general-purpose CPUs and other (frequently software-based) shading languages. The purpose of this extension is to expose to the OpenGL application writer an unprecedented degree of programmability in the computation of final fragment colors and depth values.

This extension provides a mechanism for defining fragment program instruction sequences for application-defined fragment programs. When in fragment program mode, a program is executed each time a fragment is produced by rasterization. The inputs for the program are the attributes (position, colors, texture coordinates) associated with the fragment and a set of constant registers. A fragment program can perform mathematical computations and texture lookups using arbitrary texture coordinates. The results of a fragment program are new color and depth values for the fragment.

This extension defines a programming model including a 4-component vector instruction set, 16- and 32-bit floating-point data types, and a relatively large set of temporary registers. The programming model also includes a condition code vector which can be used to mask register writes at run-time or kill fragments altogether. The syntax, program instructions, and general semantics are similar to those in the NV_vertex_program and NV_vertex_program2 extensions, which provide for the execution of an arbitrary program each time the GL receives a vertex.

The fragment program execution environment is designed for efficient hardware implementation and to support a wide variety of programs. By design, the entire set of existing fragment programs defined by existing OpenGL per-fragment computation extensions can be implemented using the extension's programming model.

The fragment program execution environment accesses textures via arbitrarily computed texture coordinates. As such, there is no necessary correspondence between the texture coordinates and texture maps previously lumped into a single "texture unit". This extension separates the notion of "texture coordinate sets" and "texture image units" (texture maps and associated parameters), allowing implementations with a different number of each. The initial implementation of this extension will support 8 texture coordinate sets and 16 texture image units.

Issues

What limitations exist in this extension?

RESOLVED: Very few. Programs can not exceed a maximum program length (which is no less than 1024 instructions), and can use no more than 32-64 temporary registers. Programs can not access more than one fragment attribute or program parameter (constant) per instruction, but can work around this restriction using temporaries. The number of textures that can be used by a program is limited to the number of texture image units provided by the implementation (16 in the initial implementation of this extension).

These limits are fairly high. Additionally, there is no limit on the total number of texture lookups that can be performed by a program. There is no limit on the length of a texture dependency chain -- one can write a program that performs over 1000 consecutive dependent texture lookups. There is no restrictions on dependencies between texture mapping instructions and arithmetic instructions. Texture lookups can be performed using arbitrarily computed texture coordinates. Applications can carry out their calculations with full 32-bit single precision, although two lower-precision modes are also available.

How does texture mapping work with fragment programs?

RESOLVED: This extension provides three instructions used to perform texture lookups.

The "TEX" instruction performs a lookup with the (s,t,r) values taken from an interpolated texture coordinate, an arbitrarily computed vector, or even a program constant. The "TXP" instruction performs a similar lookup, except that it uses the fourth component of the source vector to performs a perspective divide, using (s/q, t/q, r/q). In both cases, the GL will automatically compute partial derivatives used for filter and LOD selection.

The "TXD" instruction operates like "TEX", except that it allows the program to explicitly specify two additional vectors containing the partial derivatives of the texture coordinate with respect to x and y window coordinates.

All three instructions write a filtered texel value to a temporary or output register. Other than the computation of texture coordinates and partial derivatives, texture lookups not performed any differently in fragment program mode. In particular, any applicable LOD biases, wrap modes, minification and magnification filters, and anisotropic filtering controls are still applied in fragment program mode.

The results of the texture lookup are available to be used arbitrarily by subsequent fragment program instructions. Fragment programs are allowed to access any texture map arbitrarily many times.

Can fragment programs be used to compute depth values?

RESOLVED: Yes. A fragment program can perform arbitrary computations to compute a final value for the fragment, which it should write to the "z" component of the o[DEPR] register. The "z" value written should be in the range [0,1], regardless of the size of the depth buffer.

To assist in the computation of the final Z value, a fragment program can access the interpolated depth of the fragment (prior to any displacement) by reading the "z" component of the f[WPOS] attribute register.

How should near and far plane clipping work in fragment program mode if the current fragment program computes a depth value?

RESOLVED: Geometric clipping to the near and far clip plane should be disabled. Clipping should be done based on the depth values computed per-fragment. The rationale is that per-fragment depth displacement operations may effectively move portions of a primitive initially outside the clip volume inside, and vice versa.

Note that under the NV_depth_clamp extension, geometric clipping to the near and far clip planes is also disabled, and the fragment depth values are clamped to the depth range. If depth clamp mode is enabled when using a fragment program that computes a depth value, the computed depth value will be clamped to the depth range.

Should fragment programs be allowed to use multiple precisions for operands and operations?

RESOLVED: Yes. Low-precision operands are generally adequate for representing colors. Allowing low-precision registers also allows for a larger number of temporary registers (at lower precision). Low-precision operations also provide the opportunity for a higher level of performance.

Applications are free to use only high-precision operations or mix high- and low-precision operations as necessary.

What levels of precision are supported in arithmetic operations?

RESOLVED: Arithmetic operations can be performed at three different precisions. 32-bit floating point precision (fp32) uses the IEEE single-precision standard with a sign bit, 8 exponent bits, and 23 mantissa bits. 16-bit floating-point precision (fp16) uses a similar floating-point representation, but with 5 exponent bits and 10 mantissa bits. Additionally, many arithmetic operations can also be carried out at 12-bit fixed point precision (fx12), where values in the range [-2,+2) are represented as signed values with 10 fraction bits.

How should the precision with which operations are carried out be specified? Should we infer the precision from the types of the operands or result vectors? Or should it be an attribute of the instruction?

RESOLVED: Applications can optionally specify the precision of individual instructions by adding a suffix of "R", "H", and "X" to instruction names to select fp32, fp16, and fx12 precision, respectively.

By default, instructions will be carried out using the precision of the destination register. Always inferring the precision from the operands has a number of issues. First, there are a number of operations (e.g., TEX/TXP/TXD) where result type has little to no correspondance to the type of the operands. In these cases, precision suffixes are not supported. Second, one could have instructions automatically cast operands and compute results using the type of the highest precision operand or result. This behavior would be problematic since all fragment attribute registers and program parameters are kept at full precision, but full precision may not be needed by the operation.

The choice of precision level allows programs to trade off precision for potentially higher performance. Giving the program explicit control over the precision also allows it to dictate precision explicitly and eliminate any uncertainty over type casting.

For instructions whose specified precision is different than the precision of the operands or the result registers, how are the operations performed? How are the condition codes updated?

RESOLVED: Operations are performed with operands and results at the precision specified by the instruction. After the operation is complete, the result is converted to the precision of the destination register, after which the condition code is generated.

In an alternate approach, the condition code could be generated from the result. However, in some cases, the register contents would not match the condition code. In such cases, it may not be reliable to use the condition code to prevent division by zero or other special cases.

How does this extension interact with the ARB_multisample extension? In the ARB_multisample extension, each fragment has multiple depth values. In this extension, a single interpolated depth value may be modified by a fragment program.

RESOLVED: The depth values for the extra samples are generated by computing partials of the computed depth value and using these partials to derive the depth values for each of the extra samples.

How does this extension interact with polygon offset? Both extensions modify fragment depth values.

RESOLVED: As in the base OpenGL spec, the depth offset generated by polygon offset is added during polygon rasterization. The depth value provided to programs in f[WPOS].z already includes polygon offset, if enabled. If the depth value is replaced by a fragment program, the polygon offset value will NOT be recomputed and added back after program execution.

This is probably not desirable for fragment programs that modify depth values since the partials used to generate the offset may not match the partials of the computed depth value. Polygon offset for filled polygons can be approximated in a fragment program using the depth partials obtained by the DDX and DDY instructions. This will not work properly for line- and point-mode polygons, since the partials used for offset are computed over the polygon, while the partials resulting from the DDX and DDY instructions are computed along the line (or are zero for point-mode polygons). In addition, separate treatment of points, line segments, and polygons is not possible in a fragment program.

Should depth component replacement be an property of the fragment program or a separate enable?

RESOLVED: It should be a program property. Using the output register notation simplifies matters: depth components are replaced if and only if the DEPR register is written to. This alleviates the application and driver burden of maintaining separate state.

How does this extension affect the handling of q texture coordinates in the OpenGL spec?

RESOLVED: Fragment programs are allowed to access an associated q texture coordinate, so this attribute must be produced by rasterization. In unextended OpenGL 1.2, the q coordinate is eliminated in the rasterization portions of the spec after dividing each of s, t, and r by it. This extension updates the specification to pass q coordinates through at least to conventional texture mapping. When fragment program mode are disabled, q coordinates will be eliminated there in an identical manner. This modification has the added benefit of simplifying the equations used for attribute interpolation.

How should clip w coordinates be handled by this extension?

RESOLVED: Fragment programs are allowed to access the reciprocal of the clip w coordinate, so this attribute must be produced by rasterization. The OpenGL 1.2 spec doesn't explictly enumerate the attributes associated with the fragment, but we add treatment of the w clip coordinate in the appropriate locations.

The reciprocal of the clip w coordinate in traditional graphics hardware is produced by screen-space linear interpolation of the reciprocals of the clip w coordinates of the vertices. However, this spec says the clip w coordinate is produced by perspective-correct interpolation of the (non-reciprocated) clip w vertex coordinates. These two formulations turn out to be equivalent, and the latter is more convenient since the core OpenGL spec already contains formulas for perspective-correct interpolation of vertex attributes.

What is produced by the TEX/TXP/TXD instructions if the requested texture image is inconsistent?

RESOLVED: The result vector is specified to be (0,0,0,0). This behavior is consistent with the NV_texture_shader extension. Note that like in NV_texture_shader, these instructions ignore the standard hierarchy of texture enables and programs can access textures that are not specifically "enabled".

Should a minimum precision be specified for certain fragment attribute registers (in particular COL0, COL1) that may not be generated with full fp32 precision?

RESOLVED: No. It is expected that the precision of COL0/COL1 should generally be at least as high as that of the frame buffer.

Fragment color components (f[COL0] and f[COL1]) are generally low-precision fixed-point values in the range [0,1]. Is it possible to pass unclamped or high-precision color components to fragment programs?

RESOLVED: Yes, although you can't exactly call them "colors". High-precision per-vertex color values can be written into any unused texture coordinate set, either via a MultiTexCoord call or using a vertex program. These "texture coordinates" will be interpolated during rasterization, and can be used arbitrarily by a fragment program.

In particular, there is no requirement that per-fragment attributes called "texture coordinates" be used for texture mapping.

Should this specification guarantee that temporary registers are initialized to zero?

RESOLVED: Yes. This will allow for the modular construction of programs that accumulate results in registers. For example, per-fragment lighting may use MAD instructions to accumulate color contributions at each light. Without zero-initialization, the program would require an explicit MOV instruction to load 0 or the use of the MUL instruction for the first light.

Should this specification support Unicode program strings?

RESOLVED: Not necessary.

Programs defined by NV_vertex_program begin with "!!VP1.0". Should fragment programs have a similar identifier?

RESOLVED: Yes, "!!FP1.0", identifying the first revision of this fragment program language.

Should per-fragment attributes have equivalent integer names in the program language, as per-vertex attributes do in NV_vertex_program?

RESOLVED: No. In NV_vertex_program, "generic" vertex attributes could be specified directly by an application using only an attribute number. Those numbers may have no necessary correlation with the conventional attribute names, although conventional vertex attributes are mapped to attribute numbers. However, conventional attributes are the only outputs of vertex programs and of rasterization. Therefore, there is no need for a similar input-by-number functionality for fragment programs.

Should we provide the ability to issue instructions that do not update temporary or output registers?

RESOLVED: Yes. Programs may issue instructions whose only purpose is to update the condition code register, and requiring such instructions to write to a temporary may require the use of an additional temporary and/or defeat possible program optimizations. We accomplish this by adding two write-only temporary pseudo-registers ("RC" and "HC") that can be specified as destination registers.

Do the packing and unpacking instructions in this extension make any sense?

RESOLVED: Yes. They are useful for packing and unpacking multiple components in a single channel of a floating-point frame buffer. For example, a 128-bit "RGBA" frame buffer could pack 16 8-bit quantities or 8 16-bit quantities, all of which could be used in later rasterization passes. See the NV_float_buffer extension for more information.

Should we provide a method for specifying an fp16 depth component output value?

RESOLVED: No. There is no good reason for supporting half-precision Z outputs. Even with 16-bit Z buffers, the 10-bit mantissa of the half-precision float is rather limiting. There would effectively be only 11 good bits in the back half of the Z buffer.

Should RequestResidentProgramsNV (or a new equivalent function) take a target? Dealing with working sets of different program types is a bit messy. Should we document some limitation if we get programs of different types?

RESOLVED: In retrospect, it may have been a good idea to attach a target to this command, but there isn't a good reason to mess with something that already works for vertex programs. The driver is responsible for ensuring consistent results when the program types specified are mixed.

What happens on data type conversions where the original value is not exactly representable in the new data type, either due to overflow or insufficient precision in the destination type?

RESOLVED: In case of overflow, the original value is clamped to the +/-INF (fp16 or fp32) or the nearest representable value (fx12). In case of imprecision, the conversion is either to round or truncate to the nearest representable value.

Should this extension support IEEE-style denorms? For 32-bit IEEE floating point, denorms are numbers smaller in absolute value than 2^-126. For 16-bit floats used by this extension, denorms are numbers smaller in absolute value than 2^-14.

RESOLVED: For 32-bit data types, hardware support for denorms was considered too expensive relative to the benefit provided. Computational results that would otherwise produce denorms are flushed to zero. For 16-bit data types, hardware denorm support will be present. The expense of hardware denorm support is lower and the potential precision benefit is greater for 16-bit data types.

OpenGL provides a hierarchy of texture enables. The texture lookup operations in NV_texture_shader effectively override the texture enable hierarchy and select a specific texture to enable. What should be done by this extension?

RESOLVED: This extension will build upon NV_texture_shader and reduce the driver overhead of validating the texture enables. Texture lookups can be specified by instructions like "TEX H0, f[TEX2], TEX2, 3D", which would indicate to use texture coordinate set number 2 to do a lookup in the texture object bound to the TEXTURE_3D target in texture image unit 2.

Each texture unit can have only one "active" target. Programs are not allowed to reference different texture targets in the same texture image unit. In the example above, any other texture instructions using texture image unit 2 must specify the 3D texture target.

What is the interaction with NV_register_combiners?

RESOLVED: Register combiners are not available when fragment programs are enabled.

Previous version of this specification supported the notion of combiner programs, where the result of fragment program execution was a set of four "texture lookup" values that fed the register combiners.

For convenience, should we include pseudo-instructions not present in the hardware instruction set that are trivially implementable? For example, absolute value and subtract instructions could fall in this category. An "ABS R1,R0" instruction would be equivalent to "MAX R1,R0,-R0", and a "SUB R2,R0,R1" would be equivalent to "ADD R2,R0,-R1"

RESOLVED: In general, yes. A SUB instruction is provided for convenience. This extension does not provide a separate ABS instruction because it supports absolute value operations of each operand.

Should there be a '+' in the <optionalSign> portion of the grammar? There isn't one in the GL_NV_vertex_program spec.

RESOLVED: Yes, for orthogonality/readability. A '+' obviously adds no functionality. In NV_vertex_program, an <optionalSign> of "-" was always a negation operator. However, in fragment programs, it can also be used as a sign for a constant value.

Can the same fragment attribute register, program parameter register, or constants be used for multiple operands in the same instruction? If so, can it be used with different swizzle patterns?

RESOLVED: Yes and yes.

This extension allows different limits for the number of texture coordinate sets and the number of texture image units (i.e., texture maps and associated data). The state in ActiveTextureARB affects both *coordinate sets (TexGen, matrix operations) and image units (TexParameter, TexEnv). How should we deal with this?*

RESOLVED: Continue to use ActiveTextureARB and emit an INVALID_OPERATION if the active texture refers to an unsupported coordinate set/image unit. Other options included creating dummy (unusable) state for unsupported coordinate sets/image units and continue to use ActiveTextureARB normally, or creating separate state and state-setting commands for coordinate sets and image units. Separate state is the cleanest solution, but would add more calls and potentially cause more programmer confusion. Dummy state would avoid additional error checks, but the demands of dummy state could grow if the number of texture image units and texture coordinate sets increases.

The current OpenGL spec is vague as to what state is affected by the active texture selector and has no distination between coordinate-related and image-related state. The state tables could use a good clean-up in this area.

*The LRP instruction is defined so that the result of "LRP R0, R0, R1, R2" is R0*R1+(1-R0)*R2. There are conflicting precedents here. The definition here matches the "lrp" instruction in the DirectX 8.0 pixel shader language. However, an equivalent RenderMan lerp operation would yield a result of (1-R0)*R1+R0*R2. Which ordering should be implemented?*

RESOLVED: NVIDIA hardware implements the former operand ordering, and there is no good reason to specify a different ordering. To convert a "LRP" using the latter ordering to NV_fragment_program, swap the third and fourth arguments.

Should this extension provide tracking of matrices or any other state, similar to that provided in NV_vertex_program?

RESOLVED: No.

Should this extension provide global program parameters -- values shared between multiple fragment programs?

RESOLVED: No.

Should this extension provide program parameters specific to a program? If so, how?

RESOLVED: Yes. These parameters will be called "local parameters". This extension will provide both named and numbered local parameters. Local parameters can be managed by the driver and eliminate the need for applications to manage a global name space.

Named local parameters work much like standard variable names in most programming languages. They are created using the "DECLARE" instruction within the fragment program itself. For example:

```
DECLARE color = {1,0,0,1};
```

Named local parameters are used simply by referencing the variable name. They do not require the array syntax like the global parameters in the NV_vertex_program extension. They can be updated using the commands ProgramNamedParameter4[f,fv]NV.

Numbered local parameters are not declared. They are used by simply referencing an element of an array called "p". For example,

```
    MOV R0, p[12];
```

loads the value of numbered local parameter 12 into register R0. Numbered local parameters can be updated using the commands ProgramLocalParameter4[d,dv,f,fv]ARB.

The numbered local parameter APIs were added to this extension late in its development, and are provided for compatibility with the ARB_vertex_program extension, and what will likely be supported in ARB_fragment_program as well. Providing this mechanism allows programs to use the same mechanisms to set local parameters in both extension.

Why are the APIs for setting named and numbered local parameters different?

RESOLVED: The named parameter API was created prior to ARB_vertex_program (and the possible future ARB_fragment_program) and uses conventions borrowed from NV_vertex_program. A slightly different API was chosen during the ARB standardization process; see the ARB_vertex_program specification for more details.

The named parameter API takes a program ID and a parameter name, and sets the parameter for the program with the specified ID. The specified program does not need to be bound (via BindProgramNV) in order to modify the values of its named parameters. The numbered parameter API takes a program target enum (FRAGMENT_PROGRAM_NV) and a parameter number and modifies the corresponding numbered parameter of the currently bound program.

What should be the initial value of uninitialized local parameters?

RESOLVED: (0,0,0,0). This choice is somewhat arbitrary, but matches previous extensions (e.g., NV_vertex_program).

Should this extension support program parameter arrays?

RESOLVED: No hardware support is present. Note that from the point of view of a fragment program, a texture map can be used as a 1-, 2-, or 3-dimensional array of constants.

Should this extension provide support constants in fragment programs? If so, how?

RESOLVED: Yes. Scalar or vector constants can be defined inline (e.g., "1.0" or "{1,2,3,4}"). In addition, named constants are supported using the "DEFINE" instruction, which allow programmers to change the values of constants used in multiple instructions simply be changing the value assigned to the named constant.

Note that because this extension uses program strings, the floating-point value of any constants generated on the fly must be printed to the program string. An alternate method that avoids the need to print constants is to declare a named local program parameter and initialize it with the ProgramNamedParameter4[f,fv]() calls.

Should named constants be allowed to be redefined?

RESOLVED: No. If you want to redefine the values of constants, you can create an equivalent named program parameter by changing the "DEFINE" keyword to "DECLARE".

Should functions used to update or query named local parameters take a zero-terminated string (as with most strings in the C programming language), or should they require an explicit string length? If the former, should we create a version of LoadProgramNV that does not require a string length.

RESOLVED: Stick with explicit string length. Strings that are defined as constants can have the length computed at compile-time. Strings read from files will have the length known in advance. Programs to build strings at run-time also likely keep the length up-to-date. Passing an explicit length saves time, since the driver doesn't have to do a strlen().

What is the deal with the alpha of the secondary color?

RESOLVED: In unextended OpenGL 1.2, the alpha component of the secondary color is forced to 0.0. In the EXT_secondary_color extension, the alpha of the per-vertex secondary colors is defined to be 0.0. NV_vertex_program allows vertex programs to produce a per-vertex alpha component, but it is forced to zero for the purposes of the color sum. In the NV_register_combiners extension, the alpha component of the secondary color is undefined. What a mess.

In this extension, the alpha of the secondary color is well-defined and can be used normally. When in vertex program mode

Why are fragment program instructions involving f[FOGC] or f[TEX0] through f[TEX7] automatically carried out at full precision?

RESOLVED: This is an artifact of the method that these interpolants are generated the NVIDIA graphics hardware. If such instructions absolutely must be carried out at lower precision, the requirement can be met by first loading the interpolants into a temporary register.

With a different number of texture coordinate sets and texture image units, how many copies of each kind of texture state are there?

RESOLVED: The intention is that texture state be broken into three groups. (1) There are MAX_TEXTURE_COORDS_NV copies of texture coordinate set state, which includes current texture coordinates, TexGen state, and texture matrices. (2) There are MAX_TEXTURE_IMAGE_UNITS_NV copies of texture image unit state, which include texture maps, texture parameters, LOD bias parameters. (3) There are MAX_TEXTURE_UNITS_ARB copies of legacy OpenGL texture unit state (e.g., texture enables, TexEnv blending state), all of which are unused when in fragment program mode.

It is not necessary that MAX_TEXTURE_UNITS_ARB be equal to the minimum of MAX_TEXTURE_COORDS_NV and MAX_TEXTURE_IMAGE_UNITS -- implementations may choose not to extend fixed-function OpenGL texture mapping modes beyond a certain point.

The GLX protocol for LoadProgramNV (and ProgramNamedParameterNV) may end up with programs >64KB. This will overflow the limits of the GLX Render protocol, resulting in the need to use RenderLarge path. This is an issue with vertex programs, also.

RESOLVED: Yes, it is.

Should textures used by fragment programs be declared? For example, "TEXTURE TEX3, 2D", indicating that the 2D texture should be used for all accesses to texture unit 3. The dimension could be dropped from the TEX family of instructions, and some of the compile-time error checking could be dropped.

RESOLVED: Maybe it should be, but for better or worse, it isn't.

It is not all that uncommon to have negative q values with projective texture mapping, but results are undefined if any q values are negative in this specification. Why?

RESOLVED: This restriction carries on a similar one in the initial OpenGL specification. The motivation for this restriction is that when interpolating, it is possible for a fragment to have an interpolated q coordinate at or near 0.0. Since the texture coordinates used for projective texture mapping are s/q, t/q, and r/q, this will result in a divide-by-zero error or suffer from significant numerical instability. Results will be inaccurate for such fragments.

Other than the numerical stability issue above, NVIDIA hardware should have no problems with negative q coordinates.

Should programs that replace depth have their own special program type, Such as "!!FPD1.0" and "!!FPDC1.0"?

RESOLVED: No. If a program has an instruction that writes to o[DEPR], the final fragment depth value is taken from o[DEPR].z. Otherwise, the fragment's original depth value is used.

What fx12 value should NaN map to?

RESOLVED: For the lack of any better choice, 0.0.

How are special-case encodings (-INF, +INF, -0.0, +0.0, NaN) handled for arithmetic and comparison operations?

RESOLVED: The special cases for all floating-point operations are designed to match the IEEE specification for floating-point numbers as closely as possible. The results produced by special cases should be enumerated in the sections of this spec describing the operations. There are some cases where the implemented fragment program behavior does not match IEEE conventions, and these cases should be noted in this specification.

How can condition codes be used to mask out register writes? How about killing fragments? What other things can you do?

RESOLVED: The following example computes a component wise |R1-R2|:

```
  SUBC R0, R1, R2;      # "C" suffix means update condition code
  MOV  R0 (LT), -R0;    # Conditional write mask in parentheses
```

The first instruction computes a component-wise difference between R1 and R2, storing R1-R2 in register R0. The "C" suffix in the instruction means to update the condition code based on the sign of the result vector components. The second instruction inverts the sign of the components of R0. However the "(LT)" portion says that the destination register should be updated only if the corresponding condition code component is LT (negative). This means that only those components of R0

To kill a fragment if the red (x) component of a texture lookup returns zero:

```
  TEXC R0, f[TEX0], TEX0, 2D;
  KIL EQ.x;
```

To kill based on the green (y) component, use "EQ.y" instead. To kill if any of the four components is zero, use "EQ.xyzw" or just "EQ".

Fragment programs do not support boolean expressions. These can generally be achieved using conditional write mask.

To evaluate the expression "(R0.x == 0) && (R1.x == 0)":

```
  MOVC RC.x, R0.x;
  MOVC RC.x (EQ), R1.x;
```

To evaluate the expression "(R0.x == 0) || (R1.x == 0)":

```
  MOVC RC.x, R0.x;
  MOVC RC.x (NE), R1.x;
```

In both cases, the x component of the condition code will contain "EQ" if and only if the condition is TRUE.

How can fragment programs be used to implement non-standard texture filtering modes?

RESOLVED: As one example, consider a case where you want to do linear filtering in a 2D texture map, but only horizontally. To achieve this, first set the texture filtering mode to NEAREST. For a 16 x n texture, you might do something like:

```
  DEFINE halfTexel = { 0.03125, 0 };   # 1/32 (1/2 a texel)
  ADD R0, f[TEX0], -halfTexel;         # coords of left sample
  ADD R1, f[TEX0], +halfTexel;         # coords of right sample
  TEX R0, R0, TEX0, 2D;                # lookup left sample
  TEX R1, R1, TEX0, 2D;                # lookup right sample
  MUL R2.x, R0.x, 16;                  # scale X coords to texels
```

```
        FRC R2.x, R2.x;                    # get fraction, filter weight
        LRP R0, R2, R1, R0;                # blend samples based on weight
```

There are plenty of other interesting things that can be done.

Should this specification provide more examples?

RESOLVED: Yes, it should.

Is the OpenGL ARB working on a multi-vendor standard for fragment programmability? Will there be an ARB_fragment_program extension? If so, how will this extension interact with the ARB standard?

RESOLVED: Yes, as of July 2002, there was a multi-vendor working group and a draft specification. The ARB extension is expected to have several features not present in this extension, such as state tracking and global parameters (called "program environment parameters"). It will also likely lack certain features found in this extension.

Why does the HEMI mapping apply to the third component of signed HILO textures, but not to unsigned HILO textures?

RESOLVED: This behavior matches the behavior of NV_texture_shader (e.g., the DOT_PRODUCT_NV mode). The HEMI mapping will construct the third component of a unit vector whose first two components are encoded in the HILO texture.

New Procedures and Functions

```
    void ProgramNamedParameter4fNV(uint id, sizei len, const ubyte *name,
                                   float x, float y, float z, float w);
    void ProgramNamedParameter4dNV(uint id, sizei len, const ubyte *name,
                                   double x, double y, double z, double w);
    void ProgramNamedParameter4fvNV(uint id, sizei len, const ubyte *name,
                                    const float v[]);
    void ProgramNamedParameter4dvNV(uint id, sizei len, const ubyte *name,
                                    const double v[]);
    void GetProgramNamedParameterfvNV(uint id, sizei len, const ubyte *name,
                                      float *params);
    void GetProgramNamedParameterdvNV(uint id, sizei len, const ubyte *name,
                                      double *params);

    void ProgramLocalParameter4dARB(enum target, uint index,
                                    double x, double y, double z, double w);
    void ProgramLocalParameter4dvARB(enum target, uint index,
                                     const double *params);
    void ProgramLocalParameter4fARB(enum target, uint index,
                                    float x, float y, float z, float w);
    void ProgramLocalParameter4fvARB(enum target, uint index,
                                     const float *params);
    void GetProgramLocalParameterdvARB(enum target, uint index,
                                       double *params);
    void GetProgramLocalParameterfvARB(enum target, uint index,
                                       float *params);
```

New Tokens

Accepted by the <cap> parameter of Disable, Enable, and IsEnabled, by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev, and by the <target> parameter of BindProgramNV, LoadProgramNV, ProgramLocalParameter4dARB, ProgramLocalParameter4dvARB, ProgramLocalParameter4fARB, ProgramLocalParameter4fvARB, GetProgramLocalParameterdvARB, and GetProgramLocalParameterfvARB:

```
    FRAGMENT_PROGRAM_NV                             0x8870
```

Accepted by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

```
    MAX_TEXTURE_COORDS_NV                           0x8871
    MAX_TEXTURE_IMAGE_UNITS_NV                      0x8872
    FRAGMENT_PROGRAM_BINDING_NV                     0x8873
    MAX_FRAGMENT_PROGRAM_LOCAL_PARAMETERS_NV        0x8868
```

Accepted by the <name> parameter of GetString:

```
    PROGRAM_ERROR_STRING_NV                         0x8874
```

Additions to Chapter 2 of the OpenGL 1.2.1 Specification (OpenGL Operation)

Modify Section 2.11, Clipping (p.39)

(replace the first paragraph of the section, p. 39) Primitives are clipped to the clip volume. In clip coordinates, the view volume is defined by

```
    -w_c <= x_c <= w_c,
    -w_c <= y_c <= w_c, and
    -w_c <= z_c <= w_c.
```

Clipping to the near and far clip planes is ignored if fragment program mode (section 3.11) or texture shaders (see NV_texture_shader specification) are enabled, if the current fragment program or texture shader computes per-fragment depth values. In this case, the view volume is defined by:

```
    -w_c <= x_c <= w_c and
    -w_c <= y_c <= w_c.
```

Additions to Chapter 3 of the OpenGL 1.2.1 Specification (Rasterization)

Modify Chapter 3 introduction (p. 57)

(p.57, modify 1st paragraph) ... Figure 3.1 diagrams the rasterization process. The color value assigned to a fragment is initially determined by the rasterization operations (Sections 3.3 through 3.7) and modified by either the execution of the texturing, color sum, and fog operations as defined in Sections 3.8, 3.9, and 3.10, or of a fragment program defined in Section 3.11. The final depth value is initially determined by the rasterization operations and may be modified by a fragment program.

note: Antialiasing Application is renumbered from Section 3.11 to Section

3.12.

Modify Figure 3.1 (p.58)

```
                         Primitive Assembly
                                  |
          +-----------+-----------+-----------+-----------+
          |           |           |           |           |
          |           |           |        Pixel          |
        Point       Line       Polygon     Rectangle   Bitmap
       Raster-     Raster-     Raster-     Raster-     Raster-
       ization     ization     ization     ization     ization
          |           |           |           |           |
          +-----------+-----------+-----------+-----------+
                                  |
                                  |
                +-----------------+-----------------+
                |                 |                 |
          Conventional         Texture          Fragment
          Texture Fetch        Shaders          Programs
                |                 |                 |
                |  +--------------+                 |
                |  |                                |
    TEXTURE_    o  o                                |
    SHADER_NV                                       |
    enable      o                                   |
                |                                   |
                +-------------+                     |
                |             |                     |
           Conventional   Register                  |
              TexEnv      Combiners                 |
                |             |                     |
            Color Sum         |                     |
                |             |                     |
               Fog            |                     |
                |             |                     |
                |  +----------+                     |
                |  |                                |
    REGISTER_   o  o                                |
    COMBINERS_                                      |
    NV enable   o                                   |
                |                                   |
                +-----------------+  +--------------+
                                  |  |
                       FRAGMENT_  o  o
                       PROGRAM_
                       NV enable  o
                                  |
                                  |
                               Coverage
                              Application
                                  |
                                  v
                        to fragment processing
```

Modify Section 3.3, Points (p.61)

All fragments produced in rasterizing a non-antialiased point are assigned the same associated data, which are those of the vertex cooresponding to the point. (delete reference to divide by q).

If anitialiasing is enabled, then ... The data associated with each fragment are otherwise the data associated with the point being rasterized. (delete reference to divide by q)

Modify Section 3.4.1, Basic Line Segment Rasterization (p.66)

(Note that t=0 at p_a and t=1 at p_b). The value of an associated datum f from the fragment, whether it be R, G, B, or A (in RGBA mode) or a color index (in color index mode), the s, t, r, or q texture coordinate, or the clip w coordinate (the depth value, window z, must be found using equation 3.3, below), is found as

```
  f = (1-t) * f_a / w_a + t * f_b / w_b                    (3.2)
      --------------------------------
            (1-t) / w_a + t / w_b
```

where f_a and f_b are the data associated with the starting and ending endpoints of the segment, respectively; w_a and w_b are the clip w coordinates of the starting and ending endpoints of the segments respectively. Note that linear interpolation would use

```
  f = (1-t) * f_a + t * f_b.                               (3.3)
```

... A GL implementation may choose to approximate equation 3.2 with 3.3, but this will normally lead to unacceptable distortion effects when interpolating texture coordinates or clip w coordinates.

Modify Section 3.5.1, Basic Polygon Rasterization (p.71)

Denote a datum at p_a, p_b, or p_c ... is given by

```
  f = a * f_a / w_a + b * f_b / w_b + c * f_c / w_c        (3.4)
      ---------------------------------------------
              a / w_a + b / w_b + c / w_c
```

where w_a, w_b, and w_c are the clip w coordinates of p_a, p_b, and p_c, respectively. a, b, and c are the barycentric coordinates of the fragment for which the data are produced. a, b, and c must correspond precisely to the exact coordinates ... at the fragment's center.

Just as with line segment rasterization, equation 3.4 may be approximated by

```
  f = a * f_a + b * f_b + c * f_c;                         (3.5)
```

this may yield ... for texture coordinates or clip w coordinates.

Modify Section 3.6.4, Rasterization of Pixel Rectangles (p.100)

A fragment arising from a group ... are given by those associated with the current raster position. (delete reference to divide by q)

Modify Section 3.7, Bitmaps (p.111)

Otherwise, a rectangular array ... The associated data for each fragment are those associated with the current raster position. (delete reference to divide by q) Once the fragments have been produced ...

Modify Section 3.8, Texturing (p.112)

... an image at the location indicated by a fragment's texture coordinates to modify the fragments primary RGBA color. Texturing does not affect the secondary color.

Texturing is specified only for RGBA mode; its use in color index mode is undefined.

Except when in fragment program mode (Section 3.11), the (s,t,r) texture coordinates used for texturing are the values s/q, t/q, and r/q, respectively, where s, t, r, and q are the texture coordinates associated with the fragment. When in fragment program mode, the (s,t,r) texture coordinates are specified by the program. If q is less than or equal to zero, the results of texturing are undefined.

Add new Section 3.11, Fragment Programs (p.140)

Fragment program mode is enabled and disabled with the Enable and Disable commands using the symbolic constant FRAGMENT_PROGRAM_NV. When fragment program mode is enabled, standard and extended texturing, color sum, and fog application stages are ignored and a general purpose program is executed instead.

A fragment program is a sequence of instructions that execute on a per-fragment basis. In fragment program mode, the currently bound fragment program is executed as each fragment is generated by the rasterization operations. Fragment programs execute a finite fixed sequence of instructions with no branching or looping, and operate independently from the processing of other fragments. Fragment programs are used to compute new color values to be associated with each fragment, and can optionally compute a new depth value for each fragment as well.

Fragment program mode is not available in color index mode and is considered disabled, regardless of the state of FRAGMENT_PROGRAM_NV. When fragment program mode is enabled, texture shaders and register combiners (NV_texture_shader and NV_register_combiners extension) are disabled, regardless of the state of TEXTURE_SHADER_NV and REGISTER_COMBINERS_NV.

Section 3.11.1, Fragment Program Registers

Fragment programs operate on a set of program registers. Each program register is a 4-component vector, whose components are referred to as "x", "y", "z", and "w" respectively. The components of a fragment register are always referred to in this manner, regardless of the meaning of their contents.

The four components of each fragment program register have one of two different representations: 32-bit floating-point (fp32) or 16-bit floating-point (fp16). More details on these representations can be found in Section 3.11.4.1.

There are several different classes of program registers. Attribute registers (Table X.1) correspond to the fragment's associated data produced by rasterization. Temporary registers (Table X.2) hold intermediate results generated by the fragment program. Output registers (Table X.3) hold the final results of a fragment program. The single condition code register is used to mask writes to other registers or to determine if a fragment should be discarded.

Section 3.11.1.1, Fragment Program Attribute Registers

The fragment program attribute registers (Table X.1) hold the location of the fragment and the data associated with the fragment produced by rasterization.

Fragment Attribute Register Name	Description	Component Interpretation
f[WPOS]	Position of the fragment center.	(x,y,z,1/w)
f[COL0]	Interpolated primary color	(r,g,b,a)
f[COL1]	Interpolated secondary color	(r,g,b,a)
f[FOGC]	Interpolated fog distance/coord	(z,0,0,0)
f[TEX0]	Texture coordinate (unit 0)	(s,t,r,q)
f[TEX1]	Texture coordinate (unit 1)	(s,t,r,q)
f[TEX2]	Texture coordinate (unit 2)	(s,t,r,q)
f[TEX3]	Texture coordinate (unit 3)	(s,t,r,q)
f[TEX4]	Texture coordinate (unit 4)	(s,t,r,q)
f[TEX5]	Texture coordinate (unit 5)	(s,t,r,q)
f[TEX6]	Texture coordinate (unit 6)	(s,t,r,q)
f[TEX7]	Texture coordinate (unit 7)	(s,t,r,q)

Table X.1: Fragment Attribute Registers. The component interpretation column describes the mapping of attribute values to register components. For example, the "x" component of f[COL0] holds the red color component, and the "x" component of f[TEX0] holds the "s" texture coordinate for texture unit 0. The entries "0" and "1" indicate that the attribute register components hold the constants 0 and 1, respectively.

f[WPOS].x and f[WPOS].y hold the (x,y) window coordinates of the fragment center, and relative to the lower left corner of the window. f[WPOS].z holds the associated z window coordinate, normally in the range [0,1]. f[WPOS].w holds the reciprocal of the associated clip w coordinate.

f[COL0] and f[COL1] hold the associated RGBA primary and secondary colors of the fragment, respectively.

f[FOGC] holds the associated eye distance or fog coordinate normally used for fog computations.

f[TEX0] through f[TEX7] hold the associated texture coordinates for texture coordinate sets 0 through 7, respectively.

All attribute register components are treated as 32-bit floats. However, the components of primary and secondary colors (f[COL0] and f[COL1]) may be generated with reduced precision.

The contents of the fragment attribute registers may not be modified by a fragment program. In addition, each fragment program instruction can use at most one unique attribute register.

Section 3.11.1.2, Fragment Program Temporary Registers

The fragment temporary registers (Table X.2) hold intermediate values used during the execution of a fragment program. There are 96 temporary register names, but not all can be used simultaneously.

Fragment Temporary Register Name	Description
R0-R31	Four 32-bit (fp32) floating point values (s.e8.m23)
H0-H63	Four 16-bit (fp16) floating point values (s.e5.m10)

Table X.2: Fragment Temporary Registers.

In addition to the normal temporary registers, there are two temporary pseudo-registers, "RC" and "HC". RC and HC are treated as unnumbered, write-only temporary registers. The components of RC have an fp32 data type; the components of HC have an fp16 data type. The sole purpose of these registers is to permit instructions to modify the condition code register (section 3.11.1.4) without overwriting the values in any temporary register.

Fragment program instructions can read and write temporary registers. There is no restriction on the number of temporary registers that can be accessed by any given instruction.

All temporary registers are initialized to (0,0,0,0) each time a fragment program executes.

Section 3.11.1.3, Fragment Program Output Registers

The fragment program output registers hold the final results of the fragment program. The possible final results of a fragment program are an RGBA fragment color, a fragment depth value, and up to four texture values used by the NV_register_combiners extension.

Output Register Name	Description
o[COLR]	Final RGBA fragment color, fp32 format (color programs)
o[COLH]	Final RGBA fragment color, fp16 format (color programs)
o[TEX0]	TEXTURE0 output, fp16 format (combiner programs)
o[TEX1]	TEXTURE1 output, fp16 format (combiner programs)
o[TEX2]	TEXTURE2 output, fp16 format (combiner programs)
o[TEX3]	TEXTURE3 output, fp16 format (combiner programs)
o[DEPR]	Final fragment depth value, fp32 format

Table X.3: Fragment Program Output Registers.

o[COLR] and o[COLH] are used by color fragment programs to specify the color of a fragment. These two registers are identical, except for the associated data type of the components. The R, G, B, and A components of the fragment color are taken from the x, y, z, and w components respectively of the o[COLR] or o[COLH]. A fragment program will fail to load if it writes to both o[COLR] and o[COLH].

o[TEX0], o[TEX1], o[TEX2], and o[TEX3] are used by combiner fragment programs to generate the initial texture register values for the register combiners. After a combiner fragment program is executed, register combiner operations are performed and can use these computed values. The R, G, B, and A components of the combiner registers are taken from the x, y, z, and w components of the corresponding output registers.

o[DEPR] can be used to replace the associated depth value of a fragment. The new depth value is taken from the z component of o[DEPR]. If a fragment program does not write to o[DEPR], the associated depth value is unmodified.

A fragment program will fail to load if it does not write to at least one output register. A color fragment program will fail to load if it writes to o[TEX0], o[TEX1], o[TEX2], or o[TEX3]. A combiner fragment program will fail to load if it writes to o[COLR] or o[COLH], or if it does not write to any of o[TEX0], o[TEX1], o[TEX2], or o[TEX3].

The fragment program output registers may not be read by a fragment program, but may be written to multiple times.

The values of all fragment program output registers are initially undefined.

Section 3.11.1.4, Fragment Program Condition Code Register

The condition code register (CC) is a single four-component vector. Each component of this register is one of four enumerated values: GT (greater than), EQ (equal), LT (less than), or UN (unordered). The condition code register can be used to mask writes to fragment data register components or to terminate processing of a fragment altogether (via the KIL instruction).

Most fragment program instructions can optionally update the condition code register. When a fragment program instruction updates the condition code register, a condition code component is set to LT if the corresponding component of the result vector is less than zero, EQ if it is equal to zero, GT if it is greater than zero, and UN if it is NaN (not a number).

The condition code register is initialized to a vector of EQ values each time a fragment program executes.

Section 3.11.2, Fragment Program Parameters

In addition to using the registers defined in Section 3.11.1, fragment programs may also use fragment program parameters in their computation. Fragment program parameters are constant during the execution of fragment programs, but some parameters may be modified outside the execution of a fragment program.

There are five different types of program parameters: embedded scalar constants, embedded vector constants, named constants, named local parameters, and numbered local parameters.

Embedded scalar constants are written as standard floating-point numbers with an optional sign designator ("+" or "-") and optional scientific notation (e.g., "E+06", meaning "times 10^6").

Embedded vector constants are written as a comma-separated array of one to four scalar constants, surrounded by braces (like a C/C++ array initializer). Vector constants are always treated as 4-component vectors: constants with fewer than four components are expanded to 4-components by filling missing y and z components with 0.0 and missing w components with 1.0. Thus, the vector constant "{2}" is equivalent to "{2,0,0,1}", "{3,4}" is equivalent to "{3,4,0,1}", and "{5,6,7}" is equivalent to "{5,6,7,1}".

Named constants allow fragment program instructions to define scalar or vector constants that can be referenced by name. Named constants are created using the DEFINE instruction:

```
DEFINE pi = 3.1415926535;
DEFINE color = {0.2, 0.5, 0.8, 1.0};
```

The DEFINE instruction associates a constant name with a scalar or vector constant value. Subsequent fragment program instructions that use the constant name are equivalent to those using the corresponding constant value.

Named local parameters are similar to named vector constants, but their values can be modified after the program is loaded. Local parameters are created using the DECLARE instruction:

```
DECLARE fog_color1;
DECLARE fog_color2 = {0.3, 0.6, 0.9, 0.1};
```

The DECLARE instruction creates a 4-component vector associated with the local parameter name. Subsequent fragment program instructions referencing the local parameter name are processed as though the current value of the local parameter vector were specified instead of the parameter name. A DECLARE instruction can optionally specify an initial value for the local parameter, which can be either a scalar or vector constant. Scalar constants are expanded to 4-component vectors by replicating the scalar value in each component. The initial value of local parameters not initialized by the program is (0,0,0,0).

A named local parameter for a specific program can be updated using the calls ProgramNamedParameter4fNV or ProgramNamedParameter4fvNV (section 5.7). Named local parameters are accessible only by the program in which they are defined. Modifying a local parameter affects the only the associated program and does not affect local parameters with the same name that are found in any other fragment program.

Numbered local parameters are similar to named local parameters, except that they are referred to by number and are not declared in fragment programs. Each fragment program object has an array of four-component floating-point vectors that can be used by the program. The number of vectors is given by the implementation-dependent constant MAX_FRAGMENT_PROGRAM_LOCAL_PARAMETERS_NV, and must be at least 64. A numbered local parameter is accessed by a fragment program as members of an array called "p". For example, the instruction

```
    MOV R0, p[31];
```

copies the contents of numbered local parameter 31 into temporary register R0.

Constant and local parameter names can be arbitrary strings consisting of letters (upper or lower-case), numbers, underscores ("_"), and dollar signs ("$"). Keywords defined in the grammar (including instruction names) can not be used as constant names, nor can strings that start with numbers, or strings that specify valid temporary register or texture numbers (e.g., "R0"-"R31", "H0"-"H63"", "TEX0"-"TEX15"). A fragment program will fail to load if a DEFINE or DECLARE instruction specifies an invalid constant or local parameter name.

A fragment program will fail to load if an instruction contains a named parameter not specified in a previous DEFINE or DECLARE instruction. A fragment program will also fail to load if a DEFINE or DECLARE instruction attempts to re-define a named parameter specified in a previous DEFINE or DECLARE instruction.

The contents of the fragment program parameters may not be modified by a fragment program. In addition, each fragment program instruction can normally use at most one unique program parameter. The only exception to this rule is if all program parameter references specify named or embedded constants that taken together contain no more than four unique scalar values. For such instructions, the GL will automatically generate an equivalent instruction that references a single merged vector constant. This merging allows programs to specify instructions like the following:

```
    Instruction               Equivalent Instruction
    ---------------------     ---------------------------------------
    MAD R0, R1, 2, -1;        MAD R0, R1, {2,-1,0,0}.x, {2,-1,0,0}.y;
    ADD R0, {1,2,3,4}, 4;     ADD R0, {1,2,3,4}.xyzw, {1,2,3,4}.w;
```

Before counting the number of unique values, any named constants are first converted to the equivalent embedded constants. When generating a combined vector constant, the GL does not perform swizzling, component selection, negation, or absolute value operations. The following instructions are invalid, as they contain more than four unique scalar values.

```
    Invalid Instructions
    -----------------------------------
    ADD R0, {1,2,3,4}, -4;
    ADD R0, {1,2,3,4}, |-4|;
    ADD R0, {1,2,3,4}, -{-1,-2,-3,-4};
    ADD R0, {1,2,3,4}, {4,5,6,7}.x;
```

Section 3.11.3, Fragment Program Specification

Fragment programs are specified as an array of ubytes. The array is a string of ASCII characters encoding the program. The command LoadProgramNV loads a fragment program when the target parameter is FRAGMENT_PROGRAM_NV. The command BindProgramNV enables a fragment program for execution.

```
    At program load time, the program is parsed into a set of tokens possibly
    separated by white space.  Spaces, tabs, newlines, carriage returns, and
    comments are considered whitespace.  Comments begin with the character "#"
    and are terminated by a newline, a carriage return, or the end of the
    program array.  Fragment programs are case-sensitive -- upper and lower
    case letters are treated differently.  The proper choice of case can be
    inferred from the grammar.

    The Backus-Naur Form (BNF) grammar below specifies the syntactically valid
    sequences for fragment programs.  The set of valid tokens can be inferred
    from the grammar.  The token "" represents an empty string and is used to
    indicate optional rules.  A program is invalid if it contains any
    undefined tokens or characters.

    <program>              ::= <progPrefix> <instructionSequence> "END"

    <progPrefix>           ::= <colorProgPrefix>
                             | <combinerProgPrefix>

    <colorProgPrefix>      ::= "!!FP1.0"

    <combinerProgPrefix>   ::= "!!FCP1.0"

    <instructionSequence>  ::= <instructionSequence> <instructionStatement>
                             | <instructionStatement>

    <instructionStatement> ::= <instruction> ";"
                             | <constantDefinition> ";"
                             | <localDeclaration> ";"

    <instruction>          ::= <VECTORop-instruction>
                             | <SCALARop-instruction>
                             | <BINSCop-instruction>
                             | <BINop-instruction>
                             | <TRIop-instruction>
                             | <KILop-instruction>
                             | <TEXop-instruction>
                             | <TXDop-instruction>

    <VECTORop-instruction> ::= <VECTORop> <maskedDstReg> ","
                               <vectorSrc>
   <VECTOROp>              ::= "DDX"   | "DDX_SAT"
                             | "DDXR"  | "DDXR_SAT"
                             | "DDXH"  | "DDXH_SAT"
                             | "DDXC"  | "DDXC_SAT"
                             | "DDXRC" | "DDXRC_SAT"
                             | "DDXHC" | "DDXHC_SAT"
                             | "DDY"   | "DDY_SAT"
                             | "DDYR"  | "DDYR_SAT"
                             | "DDYH"  | "DDYH_SAT"
                             | "DDYC"  | "DDYC_SAT"
                             | "DDYRC" | "DDYRC_SAT"
                             | "DDYHC" | "DDYHC_SAT"
                             | "FLR"   | "FLR_SAT"
                             | "FLRR"  | "FLRR_SAT"
                             | "FLRH"  | "FLRH_SAT"
                             | "FLRX"  | "FLRX_SAT"
                             | "FLRC"  | "FLRC_SAT"
                             | "FLRRC" | "FLRRC_SAT"
                             | "FLRHC" | "FLRHC_SAT"
                             | "FLRXC" | "FLRXC_SAT"
                             | "FRC"   | "FRC_SAT"
                             | "FRCR"  | "FRCR_SAT"
                             | "FRCH"  | "FRCH_SAT"
                             | "FRCX"  | "FRCX_SAT"
                             | "FRCC"  | "FRCC_SAT"
                             | "FRCRC" | "FRCRC_SAT"
                             | "FRCHC" | "FRCHC_SAT"
                             | "FRCXC" | "FRCXC_SAT"
                             | "LIT"   | "LIT_SAT"
                             | "LITR"  | "LITR_SAT"
                             | "LITH"  | "LITH_SAT"
                             | "LITC"  | "LITC_SAT"
                             | "LITRC" | "LITRC_SAT"
                             | "LITHC" | "LITHC_SAT"
                             | "MOV"   | "MOV_SAT"
                             | "MOVR"  | "MOVR_SAT"
                             | "MOVH"  | "MOVH_SAT"
                             | "MOVX"  | "MOVX_SAT"
                             | "MOVC"  | "MOVC_SAT"
                             | "MOVRC" | "MOVRC_SAT"
                             | "MOVHC" | "MOVHC_SAT"
                             | "MOVXC" | "MOVXC_SAT"
                             | "PK2H"
                             | "PK2US"
                             | "PK4B"
                             | "PK4UB"

   <SCALAROp-instruction> ::= <SCALAROp> <maskedDstReg> ","
                               <scalarSrc>
    <SCALARop>              ::= "COS"     | "COS_SAT"
                             | "COSR"    | "COSR_SAT"
                             | "COSH"    | "COSH_SAT"
                             | "COSC"    | "COSC_SAT"
                             | "COSRC"   | "COSRC_SAT"
                             | "COSHC"   | "COSHC_SAT"
                             | "EX2"     | "EX2_SAT"
                             | "EX2R"    | "EX2R_SAT"
                             | "EX2H"    | "EX2H_SAT"
                             | "EX2C"    | "EX2C_SAT"
                             | "EX2RC"   | "EX2RC_SAT"
                             | "EX2HC"   | "EX2HC_SAT"
                             | "LG2"     | "LG2_SAT"
                             | "LG2R"    | "LG2R_SAT"
                             | "LG2H"    | "LG2H_SAT"
                             | "LG2C"    | "LG2C_SAT"
                             | "LG2RC"   | "LG2RC_SAT"
                             | "LG2HC"   | "LG2HC_SAT"
                             | "RCP"     | "RCP_SAT"
                             | "RCPR"    | "RCPR_SAT"
                             | "RCPH"    | "RCPH_SAT"
                             | "RCPC"    | "RCPC_SAT"
                             | "RCPRC"   | "RCPRC_SAT"
                             | "RCPHC"   | "RCPHC_SAT"
                             | "RSQ"     | "RSQ_SAT"
                             | "RSQR"    | "RSQR_SAT"
                             | "RSQH"    | "RSQH_SAT"
                             | "RSQC"    | "RSQC_SAT"
                             | "RSQRC"   | "RSQRC_SAT"
                             | "RSQHC"   | "RSQHC_SAT"
                             | "SIN"     | "SIN_SAT"
                             | "SINR"    | "SINR_SAT"
                             | "SINH"    | "SINH_SAT"
                             | "SINC"    | "SINC_SAT"
                             | "SINRC"   | "SINRC_SAT"
                             | "SINHC"   | "SINHC_SAT"
                             | "UP2H"    | "UP2H_SAT"
                             | "UP2HC"   | "UP2HC_SAT"
                             | "UP2US"   | "UP2US_SAT"
                             | "UP2USC"  | "UP2USC_SAT"
                             | "UP4B"    | "UP4B_SAT"
                             | "UP4BC"   | "UP4BC_SAT"
                             | "UP4UB"   | "UP4UB_SAT"
                             | "UP4UBC"  | "UP4UBC_SAT"

    <BINSCop-instruction> ::=  <BINSCop> <maskedDstReg> ","
                               <scalarSrc> "," <scalarSrc>

    <BINSCop>               ::= "POW"   | "POW_SAT"
                             | "POWR"  | "POWR_SAT"
                             | "POWH"  | "POWH_SAT"
                             | "POWC"  | "POWC_SAT"
                             | "POWRC" | "POWRC_SAT"
                             | "POWHC" | "POWHC_SAT"

    <BINop-instruction>     ::= <BINop> <maskedDstReg> ","
                               <vectorSrc> "," <vectorSrc>
    <BINop>                 ::= "ADD"    | "ADD_SAT"
                             | "ADDR"   | "ADDR_SAT"
                             | "ADDH"   | "ADDH_SAT"
                             | "ADDX"   | "ADDX_SAT"
                             | "ADDC"   | "ADDC_SAT"
                             | "ADDRC"  | "ADDRC_SAT"
                             | "ADDHC"  | "ADDHC_SAT"
                             | "ADDXC"  | "ADDXC_SAT"
                             | "DP3"    | "DP3_SAT"
                             | "DP3R"   | "DP3R_SAT"
                             | "DP3H"   | "DP3H_SAT"
                             | "DP3X"   | "DP3X_SAT"
                             | "DP3C"   | "DP3C_SAT"
                             | "DP3RC"  | "DP3RC_SAT"
                             | "DP3HC"  | "DP3HC_SAT"
                             | "DP3XC"  | "DP3XC_SAT"
                             | "DP4"    | "DP4_SAT"
                             | "DP4R"   | "DP4R_SAT"
                             | "DP4H"   | "DP4H_SAT"
                             | "DP4X"   | "DP4X_SAT"
                             | "DP4C"   | "DP4C_SAT"
                             | "DP4RC"  | "DP4RC_SAT"
                             | "DP4HC"  | "DP4HC_SAT"
                             | "DP4XC"  | "DP4XC_SAT"
                             | "DST"    | "DST_SAT"
                             | "DSTR"   | "DSTR_SAT"
                             | "DSTH"   | "DSTH_SAT"
                             | "DSTC"   | "DSTC_SAT"
                             | "DSTRC"  | "DSTRC_SAT"
                             | "DSTHC"  | "DSTHC_SAT"
                             | "MAX"    | "MAX_SAT"
                             | "MAXR"   | "MAXR_SAT"
                             | "MAXH"   | "MAXH_SAT"
                             | "MAXX"   | "MAXX_SAT"
                             | "MAXC"   | "MAXC_SAT"
                             | "MAXRC"  | "MAXRC_SAT"
                             | "MAXHC"  | "MAXHC_SAT"
                             | "MAXXC"  | "MAXXC_SAT"
                             | "MIN"    | "MIN_SAT"
                             | "MINR"   | "MINR_SAT"
                             | "MINH"   | "MINH_SAT"
                             | "MINX"   | "MINX_SAT"
                             | "MINC"   | "MINC_SAT"
                             | "MINRC"  | "MINRC_SAT"
                             | "MINHC"  | "MINHC_SAT"
                             | "MINXC"  | "MINXC_SAT"
                             | "MUL"    | "MUL_SAT"
                             | "MULR"   | "MULR_SAT"
                             | "MULH"   | "MULH_SAT"
                             | "MULX"   | "MULX_SAT"
                             | "MULC"   | "MULC_SAT"
                             | "MULRC"  | "MULRC_SAT"
                             | "MULHC"  | "MULHC_SAT"
                             | "MULXC"  | "MULXC_SAT"
                             | "RFL"    | "RFL_SAT"
                             | "RFLR"   | "RFLR_SAT"
```

```
                             | "RFLH"  | "RFLH_SAT"
                             | "RFLC"  | "RFLC_SAT"
                             | "RFLRC" | "RFLRC_SAT"
                             | "RFLHC" | "RFLHC_SAT"
                             | "SEQ"   | "SEQ_SAT"
                             | "SEQR"  | "SEQR_SAT"
                             | "SEQH"  | "SEQH_SAT"
                             | "SEQX"  | "SEQX_SAT"
                             | "SEQC"  | "SEQC_SAT"
                             | "SEQRC" | "SEQRC_SAT"
                             | "SEQHC" | "SEQHC_SAT"
                             | "SEQXC" | "SEQXC_SAT"
                             | "SFL"   | "SFL_SAT"
                             | "SFLR"  | "SFLR_SAT"
                             | "SFLH"  | "SFLH_SAT"
                             | "SFLX"  | "SFLX_SAT"
                             | "SFLC"  | "SFLC_SAT"
                             | "SFLRC" | "SFLRC_SAT"
                             | "SFLHC" | "SFLHC_SAT"
                             | "SFLXC" | "SFLXC_SAT"
                             | "SGE"   | "SGE_SAT"
                             | "SGER"  | "SGER_SAT"
                             | "SGEH"  | "SGEH_SAT"
                             | "SGEX"  | "SGEX_SAT"
                             | "SGEC"  | "SGEC_SAT"
                             | "SGERC" | "SGERC_SAT"
                             | "SGEHC" | "SGEHC_SAT"
                             | "SGEXC" | "SGEXC_SAT"
                             | "SGT"   | "SGT_SAT"
                             | "SGTR"  | "SGTR_SAT"
                             | "SGTH"  | "SGTH_SAT"
                             | "SGTX"  | "SGTX_SAT"
                             | "SGTC"  | "SGTC_SAT"
                             | "SGTRC" | "SGTRC_SAT"
                             | "SGTHC" | "SGTHC_SAT"
                             | "SGTXC" | "SGTXC_SAT"
                             | "SLE"   | "SLE_SAT"
                             | "SLER"  | "SLER_SAT"
                             | "SLEH"  | "SLEH_SAT"
                             | "SLEX"  | "SLEX_SAT"
                             | "SLEC"  | "SLEC_SAT"
                             | "SLERC" | "SLERC_SAT"
                             | "SLEHC" | "SLEHC_SAT"
                             | "SLEXC" | "SLEXC_SAT"
                             | "SLT"   | "SLT_SAT"
                             | "SLTR"  | "SLTR_SAT"
                             | "SLTH"  | "SLTH_SAT"
                             | "SLTX"  | "SLTX_SAT"
                             | "SLTC"  | "SLTC_SAT"
                             | "SLTRC" | "SLTRC_SAT"
                             | "SLTHC" | "SLTHC_SAT"
                             | "SLTXC" | "SLTXC_SAT"
                             | "SNE"   | "SNE_SAT"
                             | "SNER"  | "SNER_SAT"
                             | "SNEH"  | "SNEH_SAT"
                             | "SNEX"  | "SNEX_SAT"
                             | "SNEC"  | "SNEC_SAT"
```

```
                                | "SNERC" | "SNERC_SAT"
                                | "SNEHC" | "SNEHC_SAT"
                                | "SNEXC" | "SNEXC_SAT"
                                | "STR"   | "STR_SAT"
                                | "STRR"  | "STRR_SAT"
                                | "STRH"  | "STRH_SAT"
                                | "STRX"  | "STRX_SAT"
                                | "STRC"  | "STRC_SAT"
                                | "STRRC" | "STRRC_SAT"
                                | "STRHC" | "STRHC_SAT"
                                | "STRXC" | "STRXC_SAT"
                                | "SUB"   | "SUB_SAT"
                                | "SUBR"  | "SUBR_SAT"
                                | "SUBH"  | "SUBH_SAT"
                                | "SUBX"  | "SUBX_SAT"
                                | "SUBC"  | "SUBC_SAT"
                                | "SUBRC" | "SUBRC_SAT"
                                | "SUBHC" | "SUBHC_SAT"
                                | "SUBXC" | "SUBXC_SAT"

    <TRIop-instruction>    ::= <TRIop> <maskedDstReg> ","
                               <vectorSrc> "," <vectorSrc> ","
                               <vectorSrc>

    <TRIop>                ::= "MAD"   | "MAD_SAT"
                                | "MADR"  | "MADR_SAT"
                                | "MADH"  | "MADH_SAT"
                                | "MADX"  | "MADX_SAT"
                                | "MADC"  | "MADC_SAT"
                                | "MADRC" | "MADRC_SAT"
                                | "MADHC" | "MADHC_SAT"
                                | "MADXC" | "MADXC_SAT"
                                | "LRP"   | "LRP_SAT"
                                | "LRPR"  | "LRPR_SAT"
                                | "LRPH"  | "LRPH_SAT"
                                | "LRPX"  | "LRPX_SAT"
                                | "LRPC"  | "LRPC_SAT"
                                | "LRPRC" | "LRPRC_SAT"
                                | "LRPHC" | "LRPHC_SAT"
                                | "LRPXC" | "LRPXC_SAT"
                                | "X2D"   | "X2D_SAT"
                                | "X2DR"  | "X2DR_SAT"
                                | "X2DH"  | "X2DH_SAT"
                                | "X2DC"  | "X2DC_SAT"
                                | "X2DRC" | "X2DRC_SAT"
                                | "X2DHC" | "X2DHC_SAT"

    <KILop-instruction>    ::= <KILop> <ccMask>

    <KILop>                ::= "KIL"

    <TEXop-instruction>    ::= <TEXop> <maskedDstReg> ","
                               <vectorSrc> "," <texImageId>
    <TEXop>                 ::= "TEX"  | "TEX_SAT"
                              | "TEXC" | "TEXC_SAT"
                              | "TXP"  | "TXP_SAT"
                              | "TXPC" | "TXPC_SAT"

    <TXDop-instruction>     ::= <TXDop> <maskedDstReg> ","
                                <vectorSrc> "," <vectorSrc> ","
                                <vectorSrc> "," <texImageId>

    <TXDop>                 ::= "TXD"  | "TXD_SAT"
                              | "TXDC" | "TXDC_SAT"

    <scalarSrc>             ::= <absScalarSrc>
                              | <baseScalarSrc>

    <absScalarSrc>          ::= <negate> "|" <baseScalarSrc> "|"

    <baseScalarSrc>         ::= <signedScalarConstant>
                              | <negate> <namedScalarConstant>
                              | <negate> <vectorConstant> <scalarSuffix>
                              | <negate> <namedLocalParameter> <scalarSuffix>
                              | <negate> <numberedLocal> <scalarSuffix>
                              | <negate> <srcRegister> <scalarSuffix>

    <vectorSrc>             ::= <absVectorSrc>
                              | <baseVectorSrc>

    <absVectorSrc>          ::= <negate> "|" <baseVectorSrc> "|"

    <baseVectorSrc>         ::= <signedScalarConstant>
                              | <negate> <namedScalarConstant>
                              | <negate> <vectorConstant> <scalarSuffix>
                              | <negate> <vectorConstant> <swizzleSuffix>
                              | <negate> <namedLocalParameter> <scalarSuffix>
                              | <negate> <namedLocalParameter> <swizzleSuffix>
                              | <negate> <numberedLocal> <scalarSuffix>
                              | <negate> <numberedLocal> <swizzleSuffix>
                              | <negate> <srcRegister> <scalarSuffix>
                              | <negate> <srcRegister> <swizzleSuffix>

    <maskedDstReg>          ::= <dstRegister> <optionalWriteMask>
                                <optionalCCMask>

    <dstRegister>           ::= <fragTempReg>
                              | <fragOutputReg>
                              | "RC"
                              | "HC"

    <optionalCCMask>        ::= "(" <ccMask> ")"
                              | ""

    <ccMask>                ::= <ccMaskRule> <swizzleSuffix>
                              | <ccMaskRule> <scalarSuffix>

    <ccMaskRule>            ::= "EQ" | "GE" | "GT" | "LE" | "LT" | "NE" |
                                "TR" | "FL"
    <optionalWriteMask>     ::= ""
                              | "." "x"
                              | "."     "y"
                              | "." "x" "y"
                              | "."         "z"
                              | "." "x"     "z"
                              | "."     "y" "z"
                              | "." "x" "y" "z"
                              | "."             "w"
                              | "." "x"         "w"
                              | "."     "y"     "w"
                              | "." "x" "y"     "w"
                              | "."         "z" "w"
                              | "." "x"     "z" "w"
                              | "."     "y" "z" "w"
                              | "." "x" "y" "z" "w"

    <srcRegister>           ::= <fragAttribReg>
                              | <fragTempReg>

    <fragAttribReg>         ::= "f" "[" <fragAttribRegId> "]"

    <fragAttribRegId>       ::= "WPOS" | "COL0" | "COL1" | "FOGC" | "TEX0"
                              | "TEX1" | "TEX2" | "TEX3" | "TEX4" | "TEX5"
                              | "TEX6" | "TEX7"

    <fragTempReg>           ::= <fragF32Reg>
                              | <fragF16Reg>

    <fragF32Reg>            ::= "R0"  | "R1"  | "R2"  | "R3"
                              | "R4"  | "R5"  | "R6"  | "R7"
                              | "R8"  | "R9"  | "R10" | "R11"
                              | "R12" | "R13" | "R14" | "R15"
                              | "R16" | "R17" | "R18" | "R19"
                              | "R20" | "R21" | "R22" | "R23"
                              | "R24" | "R25" | "R26" | "R27"
                              | "R28" | "R29" | "R30" | "R31"

    <fragF16Reg>            ::= "H0"  | "H1"  | "H2"  | "H3"
                              | "H4"  | "H5"  | "H6"  | "H7"
                              | "H8"  | "H9"  | "H10" | "H11"
                              | "H12" | "H13" | "H14" | "H15"
                              | "H16" | "H17" | "H18" | "H19"
                              | "H20" | "H21" | "H22" | "H23"
                              | "H24" | "H25" | "H26" | "H27"
                              | "H28" | "H29" | "H30" | "H31"
                              | "H32" | "H33" | "H34" | "H35"
                              | "H36" | "H37" | "H38" | "H39"
                              | "H40" | "H41" | "H42" | "H43"
                              | "H44" | "H45" | "H46" | "H47"
                              | "H48" | "H49" | "H50" | "H51"
                              | "H52" | "H53" | "H54" | "H55"
                              | "H56" | "H57" | "H58" | "H59"
                              | "H60" | "H61" | "H62" | "H63"

    <fragOutputReg>         ::= "o" "[" <fragOutputRegName> "]"
    <fragOutputRegName>    ::= "COLR" | "COLH" | "DEPR" | "TEX0" | "TEX1"
                             | "TEX2" | "TEX3"

    <numberedLocal>        ::= "p" "[" <localNumber> "]"

    <localNumber>          ::= <integer> from 0 to
                               MAX_FRAGMENT_PROGRAM_LOCAL_PARAMETERS_NV - 1

    <scalarSuffix>         ::= "." <component>

    <swizzleSuffix>        ::= ""
                             | "." <component> <component>
                                   <component> <component>

    <component>            ::= "x" | "y" | "z" | "w"

    <texImageId>           ::= <texImageUnit> "," <texImageTarget>

    <texImageUnit>         ::= "TEX0"  | "TEX1"  | "TEX2"  | "TEX3"
                             | "TEX4"  | "TEX5"  | "TEX6"  | "TEX7"
                             | "TEX8"  | "TEX9"  | "TEX10" | "TEX11"
                             | "TEX12" | "TEX13" | "TEX14" | "TEX15"

    <texImageTarget>       ::= "1D" | "2D" | "3D" | "CUBE" | "RECT"

    <constantDefinition>   ::= "DEFINE" <namedVectorConstant> "="
                               <vectorConstant>
                             | "DEFINE" <namedScalarConstant> "="
                               <scalarConstant>

    <localDeclaration>     ::= "DECLARE" <namedLocalParameter>
                               <optionalLocalValue>

    <optionalLocalValue>   ::= ""
                             | "=" <vectorConstant>
                             | "=" <scalarConstant>

    <vectorConstant>       ::= {" <vectorConstantList> "}"
                             | <namedVectorConstant>

    <vectorConstantList>   ::= <scalarConstant>
                             | <scalarConstant> "," <scalarConstant>
                             | <scalarConstant> "," <scalarConstant> ","
                               <scalarConstant>
                             | <scalarConstant> "," <scalarConstant> ","
                               <scalarConstant> "," <scalarConstant>

    <scalarConstant>       ::= <signedScalarConstant>
                             | <namedScalarConstant>

    <signedScalarConstant> ::= <optionalSign> <floatConstant>

    <namedScalarConstant>  ::= <identifier>    ((name of a scalar constant
                                                 in a DEFINE instruction))

    <namedVectorConstant>  ::= <identifier>    ((name of a vector constant
                                                 in a DEFINE instruction))
    <namedLocalParameter>  ::= <identifier>    ((name of a local parameter
                                                 in a DECLARE instruction))

    <negate>               ::= "-" | "+" | ""

    <optionalSign>         ::= "-" | "+" | ""

    <identifier>           ::= see text below

    <floatConstant>        ::= see text below
```

The <identifier> rule matches a sequence of one or more letters ("A" through "Z", "a" through "z", "_", and "$") and digits ("0" through "9); the first character must be a letter. The underscore ("_") and dollar sign ("$") count as a letters. Upper and lower case letters are different (names are case-sensitive).

The <floatConstant> rule matches a floating-point constant consisting of an integer part, a decimal point, a fraction part, an "e" or "E", and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of on or more digits ("0" through "9"). Either the integer part or the fraction parts (not both) may be missing; either the decimal point or the "e" (or "E") and the exponent (not both) may be missing.

A fragment program fails to load if it contains more than 1024 executable instructions. Executable instructions are those matching the <instruction> rule in the grammar, and do not include DEFINE or DECLARE instructions.

A fragment program fails to load if its total temporary and output register count exceeds 64. Each fp32 temporary or output register used by the program (R0-R31, o[COLR], and o[DEPR]) counts as two registers; each fp16 temporary or output register used by the program (H0-H63 and o[COLH]) count as a single register. For combiner programs, o[TEX0], o[TEX1], o[TEX2], and o[TEX3] are counted as one register each, whether or not they are used by the program.

A fragment program fails to load if any instruction sources more than one unique fragment attribute register. Instructions sourcing the same attribute register multiple times are acceptable.

A fragment program fails to load if any instruction sources more than one unique program parameter register. Instructions sourcing the same program parameter multiple times are acceptable.

A fragment program fails to load if multiple texture lookup instructions reference different targets for the same texture image unit.

A color fragment program (indicated by the "!!FP1.0" prefix) fails to load if it writes to any of the o[TEX0], o[TEX1], o[TEX2], or o[TEX3] output registers, or if it writes to both the o[COLR] and o[COLH] output registers.

A combiner fragment program (indicated by the "!!FCP1.0" prefix) fails to load if it fails to write to any of the o[TEX0], o[TEX1], o[TEX2], or o[TEX3] output registers, or if it writes to either the o[COLR] or the o[COLH] output register.

The error INVALID_OPERATION is generated by LoadProgramNV if a fragment program fails to load because it is not syntactically correct or for one of the semantic restrictions listed above.

The error INVALID_OPERATION is generated by LoadProgramNV if a program is loaded for id when id is currently loaded with a program of a different target.

A successfully loaded fragment program is parsed into a sequence of instructions. Each instruction is identified by its tokenized name. The operation of these instructions when executed is defined in Sections 3.11.4 and 3.11.5.

Section 3.11.4, Fragment Program Operation

There are forty-five fragment program instructions. Fragment program instructions may have up to eight variants, including a suffix of "R", "H", or "X" to specify arithmetic precision (section 3.11.4.2), a suffix of "C" to allow an update of the condition code register (section 3.11.4.4), and a suffix of "_SAT" to clamp the result vector components to the range [0,1] (section 3.11.4.4). For example, the sixteen forms of the "ADD" instruction are "ADD", "ADDR", "ADDH", "ADDX", "ADDC", "ADDRC", "ADDHC", "ADDXC", "ADD_SAT", "ADDR_SAT", "ADDH_SAT", "ADDX_SAT", "ADDC_SAT", "ADDRC_SAT", "ADDHC_SAT", and "ADDXC_SAT".

Some mathematical instructions that support precision suffixes, typically those that involve complicated floating-point computations, do not support the "X" precision suffix.

The fragment program instructions and their respective input and output parameters are summarized in Table X.4.

Instruction	Inputs	Output	Description
ADD[RHX][C][_SAT]	v,v	v	add
COS[RH][C][_SAT]	s	ssss	cosine
DDX[RH][C][_SAT]	v	v	derivative relative to x
DDY[RH][C][_SAT]	v	v	derivative relative to y
DP3[RHX][C][_SAT]	v,v	ssss	3-component dot product
DP4[RHX][C][_SAT]	v,v	ssss	4-component dot product
DST[RH][C][_SAT]	v,v	v	distance vector
EX2[RH][C][_SAT]	s	ssss	exponential base 2
FLR[RHX][C][_SAT]	v	v	floor
FRC[RHX][C][_SAT]	v	v	fraction
KIL	none	none	conditionally discard fragment
LG2[RH][C][_SAT]	s	ssss	logarithm base 2
LIT[RH][C][_SAT]	v	v	compute light coefficients
LRP[RHX][C][_SAT]	v,v,v	v	linear interpolation
MAD[RHX][C][_SAT]	v,v,v	v	multiply and add
MAX[RHX][C][_SAT]	v,v	v	maximum
MIN[RHX][C][_SAT]	v,v	v	minimum
MOV[RHX][C][_SAT]	v	v	move
MUL[RHX][C][_SAT]	v,v	v	multiply
PK2H	v	ssss	pack two 16-bit floats
PK2US	v	ssss	pack two unsigned 16-bit scalars
PK4B	v	ssss	pack four signed 8-bit scalars
PK4UB	v	ssss	pack four unsigned 8-bit scalars
POW[RH][C][_SAT]	s,s	ssss	exponentiation (x^y)
RCP[RH][C][_SAT]	s	ssss	reciprocal
RFL[RH][C][_SAT]	v,v	v	reflection vector
RSQ[RH][C][_SAT]	s	ssss	reciprocal square root
SEQ[RHX][C][_SAT]	v,v	v	set on equal
SFL[RHX][C][_SAT]	v,v	v	set on false
SGE[RHX][C][_SAT]	v,v	v	set on greater than or equal
SGT[RHX][C][_SAT]	v,v	v	set on greater than
SIN[RH][C][_SAT]	s	ssss	sine
SLE[RHX][C][_SAT]	v,v	v	set on less than or equal
SLT[RHX][C][_SAT]	v,v	v	set on less than
SNE[RHX][C][_SAT]	v,v	v	set on not equal
STR[RHX][C][_SAT]	v,v	v	set on true
SUB[RHX][C][_SAT]	v,v	v	subtract
TEX[C][_SAT]	v	v	texture lookup
TXD[C][_SAT]	v,v,v	v	texture lookup w/partials
TXP[C][_SAT]	v	v	projective texture lookup
UP2H[C][_SAT]	s	v	unpack two 16-bit floats
UP2US[C][_SAT]	s	v	unpack two unsigned 16-bit scalars
UP4B[C][_SAT]	s	v	unpack four signed 8-bit scalars
UP4UB[C][_SAT]	s	v	unpack four unsigned 8-bit scalars
X2D[RH][C][_SAT]	v,v,v	v	2D coordinate transformation

Table X.4: Summary of fragment program instructions. "[RHX]" indicates an optional arithmetic precision suffix. "[C]" indicates an optional condition code update suffix. "[_SAT]" indicates an optional clamp of result vector components to [0,1]. "v" indicates a 4-component vector input or output, "s" indicates a scalar input, and "ssss" indicates a scalar output replicated across a 4-component vector.

Section 3.11.4.1: Fragment Program Storage Precision

Registers in fragment program are stored in two different representations: 16-bit floating-point (fp16) and 32-bit floating-point (fp32). There is an additional 12-bit fixed-point representation (fx12) used only as an internal representation for instructions with the "X" precision qualifier.

In the 32-bit float (fp32) representation, each component is represented in floating-point with eight exponent and twenty-three mantissa bits, as in the standard IEEE single-precision format. If S represents the sign (0 or 1), E represents the exponent in the range [0,255], and M represents the mantissa in the range [0,2^23-1], then an fp32 float is decoded as:

```
   (-1)^S * 0.0,                         if E == 0,
   (-1)^S * 2^(E-127) * (1 + M/2^23),    if 0 < E < 255,
   (-1)^S * INF,                         if E == 255 and M == 0,
   NaN,                                  if E == 255 and M != 0.
```

INF (Infinity) is a special representation indicating numerical overflow. NaN (Not a Number) is a special representation indicating the result of illegal arithmetic operations, such as division by zero. Note that all normal fp32 values, zero, and INF have an associated sign. -0.0 and +0.0 are considered equivalent for the purposes of comparisons.

This representation is identical to the IEEE single-precision floating-point standard, except that no special representation is provided for denorms -- numbers in the range (-2^-126, +2^-126). All such numbers are flushed to zero.

In a 16-bit float (fp16) register, each component is represented similarly, except with only five exponent and ten mantissa bits. If S represents the sign (0 or 1), E represents the exponent in the range [0,31], and M represents the mantissa in the range [0,2^10-1], then an fp32 float is decoded as:

```
   (-1)^S * 0.0,                         if E == 0 and M == 0,
   (-1)^S * 2^-14 * M/2^10               if E == 0 and M != 0,
   (-1)^S * 2^(E-15) * (1 + M/2^10),     if 0 < E < 31,
   (-1)^S * INF,                         if E == 31 and M == 0, or
   NaN,                                  if E == 31 and M != 0.
```

One important difference is that the fp16 representation, unlike fp32, supports denorms to maximize the limited precision of the 16-bit floating point encodings.

In the 12-bit fixed-point (fx12) format, numbers are represented as signed 12-bit two's complement integers with 10 fraction bits. The range of representable values is [-2048/1024, +2047/1024].

Section 3.11.4.2: Fragment Program Operation Precision

Fragment program instructions frequently perform mathematical operations. Such operations may be performed at one of three different precisions. Fragment programs can specify the precision of each instruction by using the precision suffix. If an instruction has a suffix of "R", calculations are carried out with 32-bit floating point operands and results. If an instruction has a suffix of "H", calculations are carried out using 16-bit floating point operands and results. If an instruction has a suffix of "X", calculations are carried out using 12-bit fixed point operands and results. For example, the instruction "MULR" performs a 32-bit floating-point multiply, "MULH" performs a 16-bit floating-point multiply, and "MULX" performs a 12-bit fixed-point multiply. If no precision suffix is specified, calculations are carried out using the precision of the temporary register receiving the result.

Fragment program instructions may source registers or constants whose precisions differ from the precision specified with the instruction. Instructions may also generate intermediate results with a different precision than that of the destination register. In these cases, the values sourced are converted to the precision specified by the instruction.

When converting to fx12 format, -INF and any values less than -2048/1024 become -2048/1024. +INF, and any values greater than +2047/1024 become +2047/1024. NaN becomes 0.

When converting to fp16 format, any values less than or equal to -2^16 are converted to -INF. Any values greater than or equal to +2^16 are converted to +INF. -INF, +INF, NaN, -0.0, and +0.0 are unchanged. Any other values that are not exactly representable in fp16 format are converted to one of the two nearest representable values.

When converting to fp32 format, any values less than or equal to -2^128 are converted to -INF. Any values greater than or equal to +2^128 are converted to +INF. -INF, +INF, NaN, -0.0, and +0.0 are unchanged. Any other values that are not exactly representable in fp32 format are converted to one of the two nearest representable values.

Fragment program instructions using the fragment attribute registers f[FOGC] or f[TEX0] through f[TEX7] will be carried out at full fp32 precision, regardless of the precision specified by the instruction.

Section 3.11.4.3: Fragment Program Operands

Except for KIL, fragment program instructions operate on either vector or scalar operands, indicated in the grammar (see section 3.11.3) by the rules <vectorSrc> and <scalarSrc> respectively.

The basic set of scalar operands is defined by the grammar rule <baseScalarSrc>. Scalar operands can be scalar constants (embedded or named), or single components of vector constants, local parameters, or registers allowed by the <srcRegister> rule. A vector component is selected by the <scalarSuffix> rule, where the characters "x", "y", "z", and "w" select the x, y, z, and w components, respectively, of the vector.

The basic set of vector operands is defined by the grammar rule <baseVectorSrc>. Vector operands can include vector constants, local parameters, or registers allowed by the <srcRegister> rule.

Basic vector operands can be swizzled according to the <swizzleSuffix> rule. In its most general form, the <swizzleSuffix> rule matches the pattern ".????" where each question mark is one of "x", "y", "z", or "w". For such patterns, the x, y, z, and w components of the operand are taken from the vector components named by the first, second, third, and fourth character of the pattern, respectively. For example, if the swizzle suffix is ".yzzx" and the specified source contains {2,8,9,0}, the swizzled operand used by the instruction is {8,9,9,2}. If the <swizzleSuffix> rule matches "", it is treated as though it were ".xyzw".

Operands can optionally be negated according to the <negate> rule in <baseScalarSrc> or <baseVectorSrc>. If the <negate> matches "-", each value is negated.

The absolute value of operands can be taken if the <vectorSrc> or <scalarSrc> rules match <absScalarSrc> or <absVectorSrc>. In this case, the absolute value of each component is taken. In addition, if the <negate> rule in <absScalarSrc> or <absVectorSrc> matches "-", the result is then negated.

Instructions requiring vector operands can also use scalar operands in the case where the <vectorSrc> rule matches <scalarSrc>. In such cases, a 4-component vector is produced by replicating the scalar.

After operands are loaded, they are converted to a data type corresponding to the operation precision specified in the fragment program instruction.

The following pseudo-code spells out the operand generation process. "SrcT" and "InstT" refer to the data types of the specified register or constant and the instruction, respectively. "VecSrcT" and "VecInstT" refer to 4-component vectors of the corresponding type. "absolute" is TRUE if the operand matches the <absScalarSrc> or <absVectorSrc> rules, and FALSE otherwise. "negateBase" is TRUE if the <negate> rule in <baseScalarSrc> or <baseVectorSrc> matches "-" and FALSE otherwise. "negateAbs" is TRUE if the <negate> rule in <absScalarSrc> or <absVectorSrc> matches "-" and FALSE otherwise. The ".c***", ".*c**", ".**c*", ".***c" modifiers refer to the x, y, z, and w components obtained by the swizzle operation. TypeConvert() is assumed to convert a scalar of type SrcT to a scalar of type InstT using the type conversion process specified above.

```
      VecInstT VectorLoad(VecSrcT source)
      {
          VecSrcT srcVal;
          VecInstT convertedVal;

          srcVal.x = source.c***;
          srcVal.y = source.*c**;
          srcVal.z = source.**c*;
          srcVal.w = source.***c;
          if (negateBase) {
             srcVal.x = -srcVal.x;
             srcVal.y = -srcVal.y;
             srcVal.z = -srcVal.z;
             srcVal.w = -srcVal.w;
          }
          if (absolute) {
             srcVal.x = abs(srcVal.x);
             srcVal.y = abs(srcVal.y);
             srcVal.z = abs(srcVal.z);
             srcVal.w = abs(srcVal.w);
          }
          if (negateAbs) {
             srcVal.x = -srcVal.x;
             srcVal.y = -srcVal.y;
             srcVal.z = -srcVal.z;
             srcVal.w = -srcVal.w;
          }

          convertedVal.x = TypeConvert(srcVal.x);
          convertedVal.y = TypeConvert(srcVal.y);
          convertedVal.z = TypeConvert(srcVal.z);
          convertedVal.w = TypeConvert(srcVal.w);
          return convertedVal;
      }

      InstT ScalarLoad(VecSrcT source)
      {
          SrcT srcVal;
          InstT convertedVal;

          srcVal = source.c***;
          if (negateBase) {
            srcVal = -srcVal;
          }
          if (absolute) {
             srcVal = abs(srcVal);
          }
          if (negateAbs) {
            srcVal = -srcVal;
          }

          convertedVal = TypeConvert(srcVal);
          return convertedVal;
      }
```

Section 3.11.4.4, Fragment Program Destination Register Update

Each fragment program instruction, except for KIL, writes a 4-component result vector to a single temporary or output register.

The four components of the result vector are first optionally clamped to the range [0,1]. The components will be clamped if and only if the result clamp suffix "_SAT" is present in the instruction name. The instruction "ADD_SAT" will clamp the results to [0,1]; the otherwise equivalent instruction "ADD" will not.

Since the instruction may be carried out at a different precision than the destination register, the components of the results vector are then converted to the data type corresponding to destination register.

Writes to individual components of the temporary register are controlled by two sets of enables: individual component write masks specified as part of the instruction and the optional condition code mask.

The component write mask is specified by the <optionalWriteMask> rule found in the <maskedDstReg> rule. If the optional mask is "", all components are enabled. Otherwise, the optional mask names the individual components to enable. The characters "x", "y", "z", and "w" match the x, y, z, and w components respectively. For example, an optional mask of ".xzw" indicates that the x, z, and w components should be enabled for writing but the y component should not. The grammar requires that the destination register mask components must be listed in "xyzw" order.

The optional condition code mask is specified by the <optionalCCMask> rule found in the <maskedDstReg> rule. If <optionalCCMask> matches "", all components are enabled. Otherwise, the condition code register is loaded and swizzled according to the swizzling specified by <swizzleSuffix>. Each component of the swizzled condition code is tested according to the rule given by <ccMaskRule>. <ccMaskRule> may have the values "EQ", "NE", "LT", "GE", LE", or "GT", which mean to enable writes if the corresponding condition code field evaluates to equal, not equal, less than, greater than or equal, less than or equal, or greater than, respectively. Comparisons involving condition codes of "UN" (unordered) evaluate to true for "NE" and false otherwise. For example, if the condition code is (GT,LT,EQ,GT) and the condition code mask is "(NE.zyxw)", the swizzle operation will load (EQ,LT,GT,GT) and the mask will thus will enable writes on the y, z, and w components. In addition, "TR" always enables writes and "FL" always disables writes, regardless of the condition code.

Each component of the destination register is updated with the result of the fragment program if and only if the component is enabled for writes by both the component write mask and the optional condition code mask. Otherwise, the component of the destination register remains unchanged.

A fragment program instruction can also optionally update the condition code register. The condition code is updated if the condition code register update suffix "C" is present in the instruction name. The instruction "ADDC" will update the condition code; the otherwise equivalent instruction "ADD" will not. If condition code updates are enabled, each component of the destination register enabled for writes is compared to zero. The corresponding component of the condition code is set to "LT", "EQ", or "GT", if the written component is less than, equal to, or greater than zero, respectively. Condition code components are set to "UN" if the written component is NaN. Note that values of -0.0 and +0.0 both evaluate to "EQ". If a component of the destination register is not enabled for writes, the corresponding condition code component is unchanged.

In the following example code,

```
    # R1=(-2, 0, 2, NaN)
    MOVC R0, R1;
    MOVC R0.xyz, R1.yzwx;
    MOVC R0 (NE), R1.zywx;
```

the first instruction writes (-2,0,2,NaN) to R0 and updates the condition code to (LT,EQ,GT,UN). The second instruction, writes to the "w" component of R0 and the condition code are disabled, so R0 ends up with (0,2,NaN,NaN) and the condition code ends up with (EQ,GT,UN,UN). In the third instruction, the condition code mask disables writes to the x component (its condition code field is "EQ"), so R0 ends up with (0,NaN,-2,0) and the condition code ends up with (EQ,UN,LT,EQ).

The following pseudocode illustrates the process of writing a result vector to the destination register. In the example, "ccMaskRule" refers to the condition code mask rule given by <ccMaskRule> (or "" if no rule is specified), "instrmask" refers to the component write mask given by the <optionalWriteMask> rule, "updatecc" is TRUE if condition code updates are enabled, and "clamp01" is TRUE if [0,1] result clamping is enabled. "destination" and "cc" refer to the register selected by <dstRegister> and the condition code, respectively.

```
  boolean TestCC(CondCode field) {
      switch (ccMaskRule) {
      case "EQ":  return (field == "EQ");
      case "NE":  return (field != "EQ");
      case "LT":  return (field == "LT");
      case "GE":  return (field == "GT" || field == "EQ");
      case "LE":  return (field == "LT" || field == "EQ");
      case "GT":  return (field == "GT");
      case "TR":  return TRUE;
      case "FL":  return FALSE;
      case "":    return TRUE;
  }

  enum GenerateCC(DstT value) {
    if (value == NaN) {
      return UN;
    } else if (value < 0) {
      return LT;
    } else if (value == 0) {
      return EQ;
    } else {
      return GT;
    }
  }
```

```
    void UpdateDestination(VecDstT destination, VecInstT result)
    {
        // Load the original destination register and condition code.
        VecDstT resultDst;
        VecDstT merged;
        VecCC   mergedCC;

        // Clamp the result vector components to [0,1], if requested.
        if (clamp01) {
            if (result.x < 0)      result.x = 0;
            else if (result.x > 1) result.x = 1;
            if (result.y < 0)      result.y = 0;
            else if (result.y > 1) result.y = 1;
            if (result.z < 0)      result.z = 0;
            else if (result.z > 1) result.z = 1;
            if (result.w < 0)      result.w = 0;
            else if (result.w > 1) result.w = 1;
        }

        // Convert the result to the type of the destination register.
        resultDst.x = TypeConvert(result.x);
        resultDst.y = TypeConvert(result.y);
        resultDst.z = TypeConvert(result.z);
        resultDst.w = TypeConvert(result.w);

        // Merge the converted result into the destination register, under
        // control of the compile- and run-time write masks.
        merged = destination;
        mergedCC = cc;
        if (instrMask.x && TestCC(cc.c***)) {
            merged.x = result.x;
            if (updatecc) mergedCC.x = GenerateCC(result.x);
        }
        if (instrMask.y && TestCC(cc.*c**)) {
            merged.y = result.y;
            if (updatecc) mergedCC.y = GenerateCC(result.y);
        }
        if (instrMask.z && TestCC(cc.**c*)) {
            merged.z = result.z;
            if (updatecc) mergedCC.z = GenerateCC(result.z);
        }
        if (instrMask.w && TestCC(cc.***c)) {
            merged.w = result.w;
            if (updatecc) mergedCC.w = GenerateCC(result.w);
        }

        // Write out the new destination register and result code.
        destination = merged;
        cc = mergedCC;
    }
```

Section 3.11.5, Fragment Program Instruction Set

The following sections describe the instruction set available to fragment programs.

Section 3.11.5.1, ADD: Add

The ADD instruction performs a component-wise add of the two operands to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = tmp0.x + tmp1.x;
  result.y = tmp0.y + tmp1.y;
  result.z = tmp0.z + tmp1.z;
  result.w = tmp0.w + tmp1.w;
```

The following special-case rules apply to addition:

1. "A+B" is always equivalent to "B+A".
2. NaN + <x> = NaN, for all <x>.
3. +INF + <x> = +INF, for all <x> except NaN and -INF.
4. -INF + <x> = -INF, for all <x> except NaN and +INF.
5. +INF + -INF = NaN.
6. -0.0 + <x> = <x>, for all <x>.
7. +0.0 + <x> = <x>, for all <x> except -0.0.

Section 3.11.5.2, COS: Cosine

The COS instruction approximates the cosine of the angle specified by the scalar operand and replicates the approximation to all four components of the result vector. The angle is specified in radians and does not have to be in the range [0,2*PI].

```
  tmp = ScalarLoad(op0);
  result.x = ApproxCosine(tmp);
  result.y = ApproxCosine(tmp);
  result.z = ApproxCosine(tmp);
  result.w = ApproxCosine(tmp);
```

The approximation function ApproxCosine is accurate to at least 22 bits with an angle in the range [0,2*PI].

```
  | ApproxCosine(x) - cos(x) | < 1.0 / 2^22, if 0.0 <= x < 2.0 * PI.
```

The error in the approximation will typically increase with the absolute value of the angle when the angle falls outside the range [0,2*PI].

The following special-case rules apply to cosine approximation:

1. ApproxCosine(NaN) = NaN.
2. ApproxCosine(+/-INF) = NaN.
3. ApproxCosine(+/-0.0) = +1.0.

Section 3.11.5.3, DDX: Derivative Relative to X

The DDX instruction computes approximate partial derivatives of the four components of the single operand with respect to the X window coordinate to yield a result vector. The partial derivative is evaluated at the center of the pixel.

```
  f = VectorLoad(op0);
  result = ComputePartialX(f);
```

Note that the partial derivates obtained by this instruction are approximate, and derivative-of-derivate instruction sequences may not yield accurate second derivatives.

For components with partial derivatives that overflow (including +/-INF inputs), the resulting partials may be encoded as large floating-point numbers instead of +/-INF.

Section 3.11.5.4, DDY: Derivative Relative to Y

The DDY instruction computes approximate partial derivatives of the four components of the single operand with respect to the Y window coordinate to yield a result vector. The partial derivative is evaluated at the center of the pixel.

```
  f = VectorLoad(op0);
  result = ComputePartialY(f);
```

Note that the partial derivates obtained by this instruction are approximate, and derivative-of-derivate instruction sequences may not yield accurate second derivatives.

For components with partial derivatives that overflow (including +/-INF inputs), the resulting partials may be encoded as large floating-point numbers instead of +/-INF.

Section 3.11.5.5, DP3: 3-Component Dot Product

The DP3 instruction computes a three component dot product of the two operands (using the x, y, and z components) and replicates the dot product to all four components of the result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1):
  result.x = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
             (tmp0.z * tmp2.z);
  result.y = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
             (tmp0.z * tmp2.z);
  result.z = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
             (tmp0.z * tmp2.z);
  result.w = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
             (tmp0.z * tmp2.z);
```

Section 3.11.5.6, DP4: 4-Component Dot Product

The DP4 instruction computes a four component dot product of the two operands and replicates the dot product to all four components of the result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1):
  result.x = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
             (tmp0.z * tmp2.z) + (tmp0.w * tmp1.w);
  result.y = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
             (tmp0.z * tmp2.z) + (tmp0.w * tmp1.w);
  result.z = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
             (tmp0.z * tmp2.z) + (tmp0.w * tmp1.w);
  result.w = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
             (tmp0.z * tmp2.z) + (tmp0.w * tmp1.w);
```

Section 3.11.5.7, DST: Distance Vector

The DST instruction computes a distance vector from two specially-formatted operands. The first operand should be of the form [NA, d^2, d^2, NA] and the second operand should be of the form [NA, 1/d, NA, 1/d], where NA values are not relevant to the calculation and d is a vector length. If both vectors satisfy these conditions, the result vector will be of the form [1.0, d, d^2, 1/d].

The exact behavior is specified in the following pseudo-code:

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = 1.0;
  result.y = tmp0.y * tmp1.y;
  result.z = tmp0.z;
  result.w = tmp1.w;
```

Given an arbitrary vector, d^2 can be obtained using the DOT3 instruction (using the same vector for both operands) and 1/d can be obtained from d^2 using the RSQ instruction.

This distance vector is useful for per-fragment light attenuation calculations: a DOT3 operation involving the distance vector and an attenuation constants vector will yield the attenuation factor.

Section 3.11.5.8, EX2: Exponential Base 2

The EX2 instruction approximates 2 raised to the power of the scalar operand and replicates it to all four components of the result vector.

```
  tmp = ScalarLoad(op0);
  result.x = Approx2ToX(tmp);
  result.y = Approx2ToX(tmp);
  result.z = Approx2ToX(tmp);
  result.w = Approx2ToX(tmp);
```

The approximation function is accurate to at least 22 bits:

```
  | Approx2ToX(x) - 2^x | < 1.0 / 2^22, if 0.0 <= x < 1.0,
```

and, in general,

```
  | Approx2ToX(x) - 2^x | < (1.0 / 2^22) * (2^floor(x)).
```

The following special-case rules apply to exponential approximation:

1. Approx2ToX(NaN) = NaN.
2. Approx2ToX(-INF) = +0.0.
3. Approx2ToX(+INF) = +INF.
4. Approx2ToX(+/-0.0) = +1.0.

Section 3.11.5.9, FLR: Floor

The FLR instruction performs a component-wise floor operation on the operand to generate a result vector. The floor of a value is defined as the largest integer less than or equal to the value. The floor of 2.3 is 2.0; the floor of -3.6 is -4.0.

```
  tmp = VectorLoad(op0);
  result.x = floor(tmp.x);
  result.y = floor(tmp.y);
  result.z = floor(tmp.z);
  result.w = floor(tmp.w);
```

The following special-case rules apply to floor computation:

1. floor(NaN) = NaN.
2. floor(<x>) = <x>, for -0.0, +0.0, -INF, and +INF. In all cases, the sign of the result is equal to the sign of the operand.

Section 3.11.5.10, FRC: Fraction

The FRC instruction extracts the fractional portion of each component of the operand to generate a result vector. The fractional portion of a component is defined as the result after subtracting off the floor of the component (see FLR), and is always in the range [0.00, 1.00).

For negative values, the fractional portion is NOT the number written to the right of the decimal point -- the fractional portion of -1.7 is not 0.7 -- it is 0.3. 0.3 is produced by subtracting the floor of -1.7 (-2.0) from -1.7.

```
  tmp = VectorLoad(op0);
  result.x = tmp.x - floor(tmp.x);
  result.y = tmp.y - floor(tmp.y);
  result.z = tmp.z - floor(tmp.z);
  result.w = tmp.w - floor(tmp.w);
```

The following special-case rules, which can be derived from the rules for FLR and ADD apply to fraction computation:

1. fraction(NaN) = NaN.
2. fraction(+/-INF) = NaN.
3. fraction(+/-0.0) = +0.0.

Section 3.11.5.11, KIL: Conditionally Discard Fragment

The KIL instruction is unlike any other instruction in the instruction set. This instruction evaluates components of a swizzled condition code using a test expression identical to that used to evaluate condition code write masks (Section 3.11.4.4). If any condition code component evaluates to TRUE, the fragment is discarded. Otherwise, the instruction has no effect. The condition code components are specified, swizzled, and evaluated in the same manner as the condition code write mask.

```
  if (TestCC(rc.c***) || TestCC(rc.*c**) ||
      TestCC(rc.**c*) || TestCC(rc.***c)) {
     // Discard the fragment.
  } else {
    // Do nothing.
  }
```

If the fragment is discarded, it is treated as though it were not produced by rasterization. In particular, none of the per-fragment operations (such as stencil tests, blends, stencil, depth, or color buffer writes) are performed on the fragment.

Section 3.11.5.12, LG2: Logarithm Base 2

The LG2 instruction approximates the base 2 logarithm of the scalar operand and replicates it to all four components of the result vector.

```
  tmp = ScalarLoad(op0);
  result.x = ApproxLog2(tmp);
  result.y = ApproxLog2(tmp);
  result.z = ApproxLog2(tmp);
  result.w = ApproxLog2(tmp);
```

The approximation function is accurate to at least 22 bits:

```
  | ApproxLog2(x) - log_2(x) | < 1.0 / 2^22.
```

The following special-case rules apply to logarithm approximation:

1. ApproxLog2(NaN) = NaN.
2. ApproxLog2(+INF) = +INF.
3. ApproxLog2(+/-0.0) = -INF.
4. ApproxLog2(x) = NaN, -INF < x < -0.0.
5. ApproxLog2(-INF) = NaN.

Section 3.11.5.13, LIT: Compute Light Coefficients

The LIT instruction accelerates per-fragment lighting by computing lighting coefficients for ambient, diffuse, and specular light contributions. The "x" component of the operand is assumed to hold a diffuse dot product (n dot VP_pli, as in the vertex lighting equations in Section 2.13.1). The "y" component of the operand is assumed to hold a specular dot product (n dot h_i). The "w" component of the operand is assumed to hold the specular exponent of the material (s_rm).

The "x" component of the result vector receives the value that should be multiplied by the ambient light/material product (always 1.0). The "y" component of the result vector receives the value that should be multiplied by the diffuse light/material product (n dot VP_pli). The "z" component of the result vector receives the value that should be multiplied by the specular light/material product (f_i * (n dot h_i) ^ s_rm). The "w" component of the result is the constant 1.0.

Negative diffuse and specular dot products are clamped to 0.0, as is done in the standard per-vertex lighting operations. In addition, if the diffuse dot product is zero or negative, the specular coefficient is forced to zero.

```
  tmp = VectorLoad(op0);
  if (t.x < 0) t.x = 0;
  if (t.y < 0) t.y = 0;
  result.x = 1.0;
  result.y = t.x;
  result.z = (t.x > 0) ? ApproxPower(t.y, t.w) : 0.0;
  result.w = 1.0;
```

The exponentiation approximation used to compute result.z are identical to that used in the POW instruction, including errors and the processing of any special cases.

Section 3.11.5.14, LRP: Linear Interpolation

The LRP instruction performs a component-wise linear interpolation to yield a result vector. It interpolates between the components of the second and third operands, using the first operand as a weight.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  tmp2 = VectorLoad(op2);
  result.x = tmp0.x * tmp1.x + (1 - tmp0.x) * tmp2.x;
  result.y = tmp0.y * tmp1.y + (1 - tmp0.y) * tmp2.y;
  result.z = tmp0.z * tmp1.z + (1 - tmp0.z) * tmp2.z;
  result.w = tmp0.w * tmp1.w + (1 - tmp0.w) * tmp2.w;
```

Section 3.11.5.15, MAD: Multiply and Add

The MAD instruction performs a component-wise multiply of the first two operands, and then does a component-wise add of the product to the third operand to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  tmp2 = VectorLoad(op2);
  result.x = tmp0.x * tmp1.x + tmp2.x;
  result.y = tmp0.y * tmp1.y + tmp2.y;
  result.z = tmp0.z * tmp1.z + tmp2.z;
  result.w = tmp0.w * tmp1.w + tmp2.w;
```

Section 3.11.5.16, MAX: maximum

The MAX instruction computes component-wise maximums of the values in the two operands to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = max(tmp0.x, tmp1.x);
  result.y = max(tmp0.y, tmp1.y);
  result.z = max(tmp0.z, tmp1.z);
  result.w = max(tmp0.w, tmp1.w);
```

The following special cases apply to the maximum operation:

1. max(A,B) is always equivalent to max(B,A).
2. max(NaN, <x>) == NaN, for all <x>.

Section 3.11.5.17, MIN: minimum

The MIN instruction computes component-wise minimums of the values in the two operands to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = min(tmp0.x, tmp1.x);
  result.y = min(tmp0.y, tmp1.y);
  result.z = min(tmp0.z, tmp1.z);
  result.w = min(tmp0.w, tmp1.w);
```

The following special cases apply to the minimum operation:

1. min(A,B) is always equivalent to min(B,A).
2. min(NaN, <x>) == NaN, for all <x>.

Section 3.11.5.18, MOV: Move

The MOV instruction copies the value of the operand to yield a result vector.

```
  result = VectorLoad(op0);
```

Section 3.11.5.19, MUL: Multiply

The MUL instruction performs a component-wise multiply of the two operands to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = tmp0.x * tmp1.x;
  result.y = tmp0.y * tmp1.y;
  result.z = tmp0.z * tmp1.z;
  result.w = tmp0.w * tmp1.w;
```

The following special-case rules apply to multiplication:

1. "A*B" is always equivalent to "B*A".
2. NaN * <x> = NaN, for all <x>.
3. +/-0.0 * +/-INF = NaN.
4. +/-0.0 * <x> = +/-0.0, for all <x> except -INF, +INF, and NaN. The sign of the result is positive if the signs of the two operands match and negative otherwise.
5. +/-INF * <x> = +/-INF, for all <x> except -0.0, +0.0, and NaN. The sign of the result is positive if the signs of the two operands match and negative otherwise.
6. +1.0 * <x> = <x>, for all <x>.

Section 3.11.5.20, PK2H: Pack Two 16-bit Floats

The PK2H instruction converts the "x" and "y" components of the single operand into 16-bit floating-point format, packs the bit representation of these two floats into a 32-bit value, and replicates that value to all four components of the result vector. The PK2H instruction can be reversed by the UP2H instruction below.

```
  tmp0 = VectorLoad(op0);
  /* result obtained by combining raw bits of tmp0.x, tmp0.y */
  result.x = RawBits(tmp0.x) | (RawBits(tmp0.y) << 16);
  result.y = RawBits(tmp0.x) | (RawBits(tmp0.y) << 16);
  result.z = RawBits(tmp0.x) | (RawBits(tmp0.y) << 16);
  result.w = RawBits(tmp0.x) | (RawBits(tmp0.y) << 16);
```

The result must be written to a register with 32-bit components (an "R" register, o[COLR], or o[DEPR]). A fragment program will fail to load if any other register type is specified.

Section 3.11.5.21, PK2US: Pack Two Unsigned 16-bit Scalars

The PK2US instruction converts the "x" and "y" components of the single operand into a packed pair of 16-bit unsigned scalars. The scalars are represented in a bit pattern where all '0' bits corresponds to 0.0 and all '1' bits corresponds to 1.0. The bit representations of the two converted components are packed into a 32-bit value, and that value is replicated to all four components of the result vector. The PK2US instruction can be reversed by the UP2US instruction below.

```
  tmp0 = VectorLoad(op0);
  if (tmp0.x < 0.0) tmp0.x = 0.0;
  if (tmp0.x > 1.0) tmp0.x = 1.0;
  if (tmp0.y < 0.0) tmp0.y = 0.0;
  if (tmp0.y > 1.0) tmp0.y = 1.0;
  us.x = round(65535.0 * tmp0.x);  /* us is a ushort vector */
  us.y = round(65535.0 * tmp0.y);
  /* result obtained by combining raw bits of us. */
  result.x = ((us.x) | (us.y << 16));
  result.y = ((us.x) | (us.y << 16));
  result.z = ((us.x) | (us.y << 16));
  result.w = ((us.x) | (us.y << 16));
```

The result must be written to a register with 32-bit components (an "R" register, o[COLR], or o[DEPR]). A fragment program will fail to load if any other register type is specified.

Section 3.11.5.22, PK4B: Pack Four Signed 8-bit Scalars

The PK4B instruction converts the four components of the single operand into 8-bit signed quantities. The signed quantities are represented in a bit pattern where all '0' bits corresponds to -128/127 and all '1' bits corresponds to +127/127. The bit representations of the four converted components are packed into a 32-bit value, and that value is replicated to all four components of the result vector. The PK4B instruction can be reversed by the UP4B instruction below.

```
  tmp0 = VectorLoad(op0);
  if (tmp0.x < -128/127) tmp0.x = -128/127;
  if (tmp0.y < -128/127) tmp0.y = -128/127;
  if (tmp0.z < -128/127) tmp0.z = -128/127;
  if (tmp0.w < -128/127) tmp0.w = -128/127;
  if (tmp0.x > +127/127) tmp0.x = +127/127;
  if (tmp0.y > +127/127) tmp0.y = +127/127;
  if (tmp0.z > +127/127) tmp0.z = +127/127;
  if (tmp0.w > +127/127) tmp0.w = +127/127;
  ub.x = round(127.0 * tmp0.x + 128.0);  /* ub is a ubyte vector */
  ub.y = round(127.0 * tmp0.y + 128.0);
  ub.z = round(127.0 * tmp0.z + 128.0);
  ub.w = round(127.0 * tmp0.w + 128.0);
  /* result obtained by combining raw bits of ub. */
  result.x = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
  result.y = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
  result.z = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
  result.w = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
```

The result must be written to a register with 32-bit components (an "R" register, o[COLR], or o[DEPR]). A fragment program will fail to load if any other register type is specified.

Section 3.11.5.23, PK4UB: Pack Four Unsigned 8-bit Scalars

The PK4UB instruction converts the four components of the single operand into a packed grouping of 8-bit unsigned scalars. The scalars are represented in a bit pattern where all '0' bits corresponds to 0.0 and all '1' bits corresponds to 1.0. The bit representations of the four converted components are packed into a 32-bit value, and that value is replicated to all four components of the result vector. The PK4UB instruction can be reversed by the UP4UB instruction below.

```
  tmp0 = VectorLoad(op0);
  if (tmp0.x < 0.0) tmp0.x = 0.0;
  if (tmp0.x > 1.0) tmp0.x = 1.0;
  if (tmp0.y < 0.0) tmp0.y = 0.0;
  if (tmp0.y > 1.0) tmp0.y = 1.0;
  if (tmp0.z < 0.0) tmp0.z = 0.0;
  if (tmp0.z > 1.0) tmp0.z = 1.0;
  if (tmp0.w < 0.0) tmp0.w = 0.0;
  if (tmp0.w > 1.0) tmp0.w = 1.0;
  ub.x = round(255.0 * tmp0.x);  /* ub is a ubyte vector */
  ub.y = round(255.0 * tmp0.y);
  ub.z = round(255.0 * tmp0.z);
  ub.w = round(255.0 * tmp0.w);
  /* result obtained by combining raw bits of ub. */
  result.x = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
  result.y = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
  result.z = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
  result.w = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
```

The result must be written to a register with 32-bit components (an "R" register, o[COLR], or o[DEPR]). A fragment program will fail to load if any other register type is specified.

Section 3.11.5.24, POW: Exponentiation

The POW instruction approximates the value of the first scalar operand raised to the power of the second scalar operand and replicates it to all four components of the result vector.

```
  tmp0 = ScalarLoad(op0);
  tmp1 = ScalarLoad(op1);
  result.x = ApproxPower(tmp0, tmp1);
  result.y = ApproxPower(tmp0, tmp1);
  result.z = ApproxPower(tmp0, tmp1);
  result.w = ApproxPower(tmp0, tmp1);
```

The exponentiation approximation function is defined in terms of the base 2 exponentiation and logarithm approximation operations in the EX2 and LG2 instructions, including errors and the processing of any special cases. In particular,

```
  ApproxPower(a,b) = ApproxExp2(b * ApproxLog2(a)).
```

The following special-case rules, which can be derived from the rules in the LG2, MUL, and EX2 instructions, apply to exponentiation:

```
  1. ApproxPower(<x>, <y>) = NaN, if x < -0.0,
  2. ApproxPower(<x>, <y>) = NaN, if x or y is NaN.
  3. ApproxPower(+/-0.0, +/-0.0) = NaN.
  4. ApproxPower(+INF, +/-0.0) = NaN.
  5. ApproxPower(+1.0, +/-INF) = NaN.
  6. ApproxPower(+/-0.0, <x>) = +0.0, if x > +0.0.
  7. ApproxPower(+/-0.0, <x>) = +INF, if x < -0.0.
  8. ApproxPower(+1.0, <x>)   = +1.0, if -INF < x < +INF.
  9. ApproxPower(+INF, <x>) = +INF, if x > +0.0.
  10. ApproxPower(+INF, <x>) = +INF, if x < -0.0.
  11. ApproxPower(<x>, +/-0.0) = +1.0, if +0.0 < x < +INF.
  12. ApproxPower(<x>, +1.0) ~= <x>, if x >= +0.0.
  13. ApproxPower(<x>, +INF) = +0.0, if -0.0 <= x < +1.0,
                               +INF, if x > +1.0,
  14. ApproxPower(<x>, -INF) = +INF, if -0.0 <= x < +1.0,
                               +0.0, if x > +1.0,
```

Note that 0^0 is defined here as NaN, since ApproxLog2(0) = -INF, and 0*(-INF) = NaN. In many other applications, including the standard C pow() function, 0^0 is defined as 1.0. This behavior can be emulated using additional instructions in much that same way that the pow() function is implemented on many CPUs.

Note that a logarithm is involved even if the exponent is an integer. This means that any exponentiating with a negative base will produce NaN. In constrast, it is possible in a "normal" mathematical formulation to raise negative numbers to integral powers (e.g., (-3)^2== 9, and (-0.5)^-2==4).

Section 3.11.5.25, RCP: Reciprocal

The RCP instruction approximates the reciprocal of the scalar operand and replicates it to all four components of the result vector.

```
  tmp = ScalarLoad(op0);
  result.x = ApproxReciprocal(tmp);
  result.y = ApproxReciprocal(tmp);
  result.z = ApproxReciprocal(tmp);
  result.w = ApproxReciprocal(tmp);
```

The approximation function is accurate to at least 22 bits:

```
  | ApproxReciprocal(x) - (1/x) | < 1.0 / 2^22, if 1.0 <= x < 2.0.
```

The following special-case rules apply to reciprocation:

```
  1. ApproxReciprocal(NaN) = NaN.
  2. ApproxReciprocal(+INF) = +0.0.
  3. ApproxReciprocal(-INF) = -0.0.
  4. ApproxReciprocal(+0.0) = +INF.
  5. ApproxReciprocal(-0.0) = -INF.
```

Section 3.11.5.26, RFL: Reflection Vector

The RFL instruction computes the reflection of the second vector operand (the "direction" vector) about the vector specified by the first vector operand (the "axis" vector). Both operands are treated as 3D vectors (the w components are ignored). The result vector is another 3D vector (the "reflected direction" vector). The length of the result vector, ignoring rounding errors, should equal that of the second operand.

```
  axis = VectorLoad(op0);
  direction = VectorLoad(op1);
  tmp.w = (axis.x * axis.x + axis.y * axis.y +
           axis.z * axis.z);
  tmp.x = (axis.x * direction.x + axis.y * direction.y +
           axis.z * direction.z);
  tmp.x = 2.0 * tmp.x;
  tmp.x = tmp.x / tmp.w;
  result.x = tmp.x * axis.x - direction.x;
  result.y = tmp.x * axis.y - direction.y;
  result.z = tmp.x * axis.z - direction.z;
```

A fragment program will fail to load if the w component of the result is enabled in the component write mask (see the <optionalWriteMask> rule in the grammar).

Section 3.11.5.27, RSQ: Reciprocal Square Root

The RSQ instruction approximates the reciprocal of the square root of the scalar operand and replicates it to all four components of the result vector.

```
  tmp = ScalarLoad(op0);
  result.x = ApproxRSQRT(tmp);
  result.y = ApproxRSQRT(tmp);
  result.z = ApproxRSQRT(tmp);
  result.w = ApproxRSQRT(tmp);
```

The approximation function is accurate to at least 22 bits:

```
  | ApproxRSQRT(x) - (1/x) | < 1.0 / 2^22, if 1.0 <= x < 4.0.
```

The following special-case rules apply to reciprocal square roots:

```
  1. ApproxRSQRT(NaN) = NaN.
  2. ApproxRSQRT(+INF) = +0.0.
  3. ApproxRSQRT(-INF) = NaN.
  4. ApproxRSQRT(+0.0) = +INF.
  5. ApproxRSQRT(-0.0) = -INF.
  6. ApproxRSQRT(x) = NaN, if -INF < x < -0.0.
```

Section 3.11.5.28, SEQ: Set on Equal To

The SEQ instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operand is equal to that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x == tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y == tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z == tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w == tmp1.w) ? 1.0 : 0.0;
```

The following special-case rules apply to SEQ:

```
  1. (<x> == <y>) and (<y> == <x>) always produce the same result.
  1. (NaN == <x>) is FALSE for all <x>, including NaN.
  2. (+INF == +INF) and (-INF == -INF) are TRUE.
  3. (-0.0 == +0.0) and (+0.0 == -0.0) are TRUE.
```

Section 3.11.5.29, SFL: Set on False

The SFL instruction is a degenerate case of the other "Set on" instructions that sets all components of the result vector to 0.0.

```
  result.x = 0.0;
  result.y = 0.0;
  result.z = 0.0;
  result.w = 0.0;
```

Section 3.11.5.30, SGE: Set on Greater Than or Equal

The SGE instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operands is greater than or equal that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x >= tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y >= tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z >= tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w >= tmp1.w) ? 1.0 : 0.0;
```

The following special-case rules apply to SGE:

1. (NaN >= <x>) and (<x> >= NaN) are FALSE for all <x>.
2. (+INF >= +INF) and (-INF >= -INF) are TRUE.
3. (-0.0 >= +0.0) and (+0.0 >= -0.0) are TRUE.

Section 3.11.5.31, SGT: Set on Greater Than

The SGT instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operands is greater than that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x > tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y > tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z > tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w > tmp1.w) ? 1.0 : 0.0;
```

The following special-case rules apply to SGT:

1. (NaN > <x>) and (<x> > NaN) are FALSE for all <x>.
2. (-0.0 > +0.0) and (+0.0 > -0.0) are FALSE.

Section 3.11.5.32, SIN: Sine

The SIN instruction approximates the sine of the angle specified by the scalar operand and replicates it to all four components of the result vector. The angle is specified in radians and does not have to be in the range [0,2*PI].

```
  tmp = ScalarLoad(op0);
  result.x = ApproxSine(tmp);
  result.y = ApproxSine(tmp);
  result.z = ApproxSine(tmp);
  result.w = ApproxSine(tmp);
```

The approximation function is accurate to at least 22 bits with an angle in the range [0,2*PI].

```
  | ApproxSine(x) - sin(x) | < 1.0 / 2^22, if 0.0 <= x < 2.0 * PI.
```

The error in the approximation will typically increase with the absolute value of the angle when the angle falls outside the range [0,2*PI].

The following special-case rules apply to cosine approximation:

1. ApproxSine(NaN) = NaN.
2. ApproxSine(+/-INF) = NaN.
3. ApproxSine(+/-0.0) = +/-0.0. The sign of the result is equal to the sign of the single operand.

Section 3.11.5.33, SLE: Set on Less Than or Equal

The SLE instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operand is less than or equal to that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x <= tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y <= tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z <= tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w <= tmp1.w) ? 1.0 : 0.0;
```

The following special-case rules apply to SLE:

1. (NaN <= <x>) and (<x> <= NaN) are FALSE for all <x>.
2. (+INF <= +INF) and (-INF <= -INF) are TRUE.
3. (-0.0 <= +0.0) and (+0.0 <= -0.0) are TRUE.

Section 3.11.5.34, SLT: Set on Less Than

The SLT instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operand is less than that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x < tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y < tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z < tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w < tmp1.w) ? 1.0 : 0.0;
```

The following special-case rules apply to SLT:

```
  1. (NaN < <x>) and (<x> < NaN) are FALSE for all <x>.
  2. (-0.0 < +0.0) and (+0.0 < -0.0) are FALSE.
```

Section 3.11.5.35, SNE: Set on Not Equal

The SNE instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operand is not equal to that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x != tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y != tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z != tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w != tmp1.w) ? 1.0 : 0.0;
```

The following special-case rules apply to SNE:

```
  1. (<x> != <y>) and (<y> != <x>) always produce the same result.
  2. (NaN != <x>) is TRUE for all <x>, including NaN.
  3. (+INF != +INF) and (-INF != -INF) are FALSE.
  4. (-0.0 != +0.0) and (+0.0 != -0.0) are TRUE.
```

Section 3.11.5.36, STR: Set on True

The STR instruction is a degenerate case of the other "Set on" instructions that sets all components of the result vector to 1.0.

```
  result.x = 1.0;
  result.y = 1.0;
  result.z = 1.0;
  result.w = 1.0;
```

Section 3.11.5.37, SUB: Subtract

The SUB instruction performs a component-wise subtraction of the second operand from the first to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = tmp0.x - tmp1.x;
  result.y = tmp0.y - tmp1.y;
  result.z = tmp0.z - tmp1.z;
  result.w = tmp0.w - tmp1.w;
```

The SUB instruction is completely equivalent to an identical ADD instruction in which the negate operator on the second operand is reversed:

```
  1. "SUB R0, R1, R2" is equivalent to "ADD R0, R1, -R2".
  2. "SUB R0, R1, -R2" is equivalent to "ADD R0, R1, R2".
  3. "SUB R0, R1, |R2|" is equivalent to "ADD R0, R1, -|R2|".
  4. "SUB R0, R1, -|R2|" is equivalent to "ADD R0, R1, |R2|".
```

Section 3.11.5.38, TEX: Texture Lookup

The TEX instruction performs a filtered texture lookup using the texture target given by <texImageTarget> belonging to the texture image unit given by <texImageUnit>. <texImageTarget> values of "1D", "2D", "3D", "CUBE", and "RECT" correspond to the texture targets TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP_ARB, and TEXTURE_RECTANGLE_NV, respectively.

The (s,t,r) texture coordinates used for the lookup are the x, y, and z components of the single operand.

The texture lookup is performed as specified in Section 3.8. The LOD calculations in Section 3.8.5 are performed using an implementation dependent method to derive ds/dx, ds/dy, dt/dx, dt/dy, dr/dx, and dr/dy. The mapping of filtered texture components to the components of the result vector is dependent on the base internal format of the texture and is specified in Table X.5.

	Result Vector Components			
Base Internal Format	X	Y	Z	W
ALPHA	0.0	0.0	0.0	At
LUMINANCE	Lt	Lt	Lt	1.0
LUMINANCE_ALPHA	Lt	Lt	Lt	At
INTENSITY	It	It	It	It
RGB	Rt	Gt	Bt	1.0
RGBA	Rt	Gt	Bt	At
HILO_NV (signed)	HIt	LOt	HEMI	1.0
HILO_NV (unsigned)	HIt	LOt	1.0	1.0
DSDT_NV	DSt	DTt	0.0	1.0
DSDT_MAG_NV	DSt	DTt	MAGt	1.0
DSDT_MAG_INTENSITY_NV	DSt	DTt	MAGt	It
FLOAT_R_NV	Rt	0.0	0.0	1.0
FLOAT_RG_NV	Rt	Gt	0.0	1.0
FLOAT_RGB_NV	Rt	Gt	Bt	1.0
FLOAT_RGBA_NV	Rt	Gt	Bt	At

Table X.5: Mapping of filtered texel components to result vector components for the TEX instruction. 0.0 and 1.0 indicate that the corresponding constant value is written to the result vector. DEPTH_COMPONENT textures are treated as ALPHA, LUMINANCE, or INTENSITY, as specified in the texture's depth texture mode.

For HILO_NV textures with signed components, "HEMI" is defined as sqrt(MAX(0, 1-(HIt^2+LOt^2))).

This instruction specifies a particular texture target, ignoring the standard hierarchy of texture enables (TEXTURE_CUBE_MAP_ARB, TEXTURE_3D, TEXTURE_2D, TEXTURE_1D) used to select a texture target in unextended OpenGL. If the specified texture target has a consistent set of images, a lookup is performed. Otherwise, the result of the instruction is the vector (0,0,0,0).

Although this instruction allows the selection of any texture target, a fragment program can not use more than one texture target for any given texture image unit.

Section 3.11.5.39, TXD: Texture Lookup with Derivatives

The TXD instruction performs a filtered texture lookup using the texture target given by <texImageTarget> belonging to the texture image unit given by <texImageUnit>. <texImageTarget> values of "1D", "2D", "3D", "CUBE", and "RECT" correspond to the texture targets TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP_ARB, and TEXTURE_RECTANGLE_NV, respectively.

The (s,t,r) texture coordinates used for the lookup are the x, y, and z components of the first operand. The partial derivatives in the X direction (ds/dx, dt/dx, dr/dx) are specified by the x, y, and z components of the second operand. The partial derivatives in the Y direction (ds/dy, dt/dy, dr/dy) are specified by the x, y, and z components of the third operand.

The texture lookup is performed as specified in Section 3.8. The LOD calculations in Section 3.8.5 are performed using the specified partial derivatives. The mapping of filtered texture components to the components of the result vector is dependent on the base internal format of the texture and is specified in Table X.5.

This instruction specifies a particular texture target, ignoring the standard hierarchy of texture enables (TEXTURE_CUBE_MAP_ARB, TEXTURE_3D, TEXTURE_2D, TEXTURE_1D) used to select a texture target in unextended OpenGL. If the specified texture target has a consistent set of images, a lookup is performed. Otherwise, the result of the instruction is the vector (0,0,0,0).

Although this instruction allows the selection of any texture target, a fragment program can not use more than one texture target for any given texture image unit.

Section 3.11.5.40, TXP: Projective Texture Lookup

The TXP instruction performs a filtered texture lookup using the texture target given by <texImageTarget> belonging to the texture image unit given by <texImageUnit>. <texImageTarget> values of "1D", "2D", "3D", "CUBE", and "RECT" correspond to the texture targets TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP_ARB, and TEXTURE_RECTANGLE_NV, respectively.

For cube map textures, the (s,t,r) texture coordinates used for the lookup are given by x, y, and z, respectively. For all other textures, the (s,t,r) texture coordinates used for the lookup are given by x/w, y/w, and z/w, respectively, where x, y, z, and w are the corresponding components of the operand.

The texture lookup is performed as specified in Section 3.8. The LOD calculations in Section 3.8.5 are performed using an implementation dependent method to derive ds/dx, ds/dy, dt/dx, dt/dy, dr/dx, and dr/dy. The mapping of filtered texture components to the components of the result vector is dependent on the base internal format of the texture and is specified in Table X.5.

This instruction specifies a particular texture target, ignoring the standard hierarchy of texture enables (TEXTURE_CUBE_MAP_ARB, TEXTURE_3D, TEXTURE_2D, TEXTURE_1D) used to select a texture target in unextended OpenGL. If the specified texture target has a consistent set of images, a lookup is performed. Otherwise, the result of the instruction is the vector (0,0,0,0).

Although this instruction allows the selection of any texture target, a fragment program can not use more than one texture target for any given texture image unit.

Section 3.11.5.41, UP2H: Unpack Two 16-Bit Floats

The UP2H instruction unpacks two 16-bit floats stored together in a 32-bit scalar operand. The first 16-bit float (stored in the 16 least significant bits) is written into the "x" and "z" components of the result vector; the second is written into the "y" and "w" components of the result vector.

This operation undoes the type conversion and packing performed by the PK2H instruction.

```
  tmp = ScalarLoad(op0);
  result.x = (fp16) (RawBits(tmp) & 0xFFFF);
  result.y = (fp16) ((RawBits(tmp) >> 16) & 0xFFFF);
  result.z = (fp16) (RawBits(tmp) & 0xFFFF);
  result.w = (fp16) ((RawBits(tmp) >> 16) & 0xFFFF);
```

Since the source operand must be a 32-bit scalar, a fragment program will fail to load if the operand is not obtained from a register with 32-bit components or from a program parameter.

Section 3.11.5.42, UP2US: Unpack Two Unsigned 16-Bit Scalars

The UP2US instruction unpacks two 16-bit unsigned values packed together in a 32-bit scalar operand. The unsigned quantities are encoded where a bit pattern of all '0' bits corresponds to 0.0 and a pattern of all '1' bits corresponds to 1.0. The "x" and "z" components of the result vector are obtained from the 16 least significant bits of the operand; the "y" and "w" components are obtained from the 16 most significant bits.

This operation undoes the type conversion and packing performed by the PK2US instruction.

```
  tmp = ScalarLoad(op0);
  result.x = ((RawBits(tmp) >> 0)  & 0xFFFF) / 65535.0;
  result.y = ((RawBits(tmp) >> 16) & 0xFFFF) / 65535.0;
  result.z = ((RawBits(tmp) >> 0)  & 0xFFFF) / 65535.0;
  result.w = ((RawBits(tmp) >> 16) & 0xFFFF) / 65535.0;
```

Since the source operand must be a 32-bit scalar, a fragment program will fail to load if the operand is not obtained from a register with 32-bit components or from a program parameter.

Section 3.11.5.43, UP4B: Unpack Four Signed 8-Bit Values

The UP4B instruction unpacks four 8-bit signed values packed together in a 32-bit scalar operand. The signed quantities are encoded where a bit pattern of all '0' bits corresponds to -128/127 and a pattern of all '1' bits corresponds to +127/127. The "x" component of the result vector is the converted value corresponding to the 8 least significant bits of the operand; the "w" component corresponds to the 8 most significant bits.

This operation undoes the type conversion and packing performed by the PK4B instruction.

```
  tmp = ScalarLoad(op0);
  result.x = (((RawBits(tmp) >> 0) & 0xFF) - 128) / 127.0;
  result.y = (((RawBits(tmp) >> 8) & 0xFF) - 128) / 127.0;
  result.z = (((RawBits(tmp) >> 16) & 0xFF) - 128) / 127.0;
  result.w = (((RawBits(tmp) >> 24) & 0xFF) - 128) / 127.0;
```

Since the source operand must be a 32-bit scalar, a fragment program will fail to load if the operand is not obtained from a register with 32-bit components or from a program parameter.

Section 3.11.5.44, UP4UB: Unpack Four Unsigned 8-Bit Scalars

The UP4UB instruction unpacks four 8-bit unsigned values packed together in a 32-bit scalar operand. The unsigned quantities are encoded where a bit pattern of all '0' bits corresponds to 0.0 and a pattern of all '1' bits corresponds to 1.0. The "x" component of the result vector is obtained from the 8 least significant bits of the operand; the "w" component is obtained from the 8 most significant bits.

This operation undoes the type conversion and packing performed by the PK4UB instruction.

```
  tmp = ScalarLoad(op0);
  result.x = ((RawBits(tmp) >> 0)  & 0xFF) / 255.0;
  result.y = ((RawBits(tmp) >> 8)  & 0xFF) / 255.0;
  result.z = ((RawBits(tmp) >> 16) & 0xFF) / 255.0;
  result.w = ((RawBits(tmp) >> 24) & 0xFF) / 255.0;
```

Since the source operand must be a 32-bit scalar, a fragment program will fail to load if the operand is not obtained from a register with 32-bit components or from a program parameter.

Section 3.11.5.45, X2D: 2D Coordinate Transformation

The X2D instruction multiplies the 2D offset vector specified by the "x" and "y" components of the second vector operand by the 2x2 matrix specified by the four components of the third vector operand, and adds the transformed offset vector to the 2D vector specified by the "x" and "y" components of the first vector operand. The first component of the sum is written to the "x" and "z" components of the result; the second component is written to the "y" and "w" components of the result.

The X2D instruction can be used to displace texture coordinates in the same manner as the OFFSET_TEXTURE_2D_NV mode in the GL_NV_texture_shader extension.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  tmp2 = VectorLoad(op2);
  result.x = tmp0.x + tmp1.x * tmp2.x + tmp1.y * tmp2.y;
  result.y = tmp0.y + tmp1.x * tmp2.z + tmp1.y * tmp2.w;
  result.z = tmp0.x + tmp1.x * tmp2.x + tmp1.y * tmp2.y;
  result.w = tmp0.y + tmp1.x * tmp2.z + tmp1.y * tmp2.w;
```

Section 3.11.6, Fragment Program Outputs

Upon completion of fragment program execution, the output registers are used to replace the fragment's associated data.

For color fragment programs, the RGBA color of the fragment is taken from the output register (COLR or COLH). The R, G, B, and A color components are extracted from the "x", "y", "z", and "w" components, respectively, of the output register and are clamped to the range [0,1].

For combiner fragment programs, register combiner operations (as described in the NV_register_combiners specification) are then performed, regardless of the state of the REGISTER_COMBINERS_NV enable. The RGBA texture colors corresponding the TEXTURE0_ARB, TEXTURE1_ARB, TEXTURE2_ARB, and TEXTURE3_ARB combiner registers are taken from the TEX0, TEX1, TEX2, and TEX3 output registers, respectively. Any components of the TEX0, TEX1, TEX2, or TEX3 output registers that are not written to by the fragment program are undefined. The R, G, B, and A texture color components are extracted from the "x", "y", "z", and "w" output register components, respectively, and are clamped to the range [-1,1].

If the DEPR output register is written by the fragment program, the depth value of the fragment is taken from the z component of the DEPR output register. If depth clamping is enabled, the depth value is clamped to the range [min(n,f), max(n,f)], where n and f are the near and far depth range values. If depth clamping is disabled, the fragment is discarded if its depth value is outside the range [min(n,f), max(n,f)].

Section 3.11.7, Required Fragment Program State

The state required for managing fragment programs consists of:

a bit indicating whether or not fragment program mode is enabled;

an unsigned integer naming the currently bound fragment program

and the state that must be maintained to indicate which integers are currently in use as fragment program names.

Fragment program mode is initially disabled. The initial state of all 128 fragment program parameter registers is (0,0,0,0). The initial currently bound fragment program is zero.

Each fragment program object consists of:

an enumerant given the program target (FRAGMENT_PROGRAM_NV);

a boolean indicating whether the program is resident;

an array of type ubyte containing the program string;

an integer representing the length of the program string array;

one four-component floating-point vector for each named local parameter in the program;

and a set of MAX_FRAGMENT_PROGRAM_LOCAL_PARAMETERS_NV four-component floating-point vectors to hold numbered local parameters, each initially set to (0,0,0,0).

Initially, no program objects exist.

Additionally, the state required during the execution of a fragment program consists of: twelve 4-component floating-point fragment attribute registers, thirty-two 128-bit physical temporary registers, and a single 4-component condition code, whose components have one of four values (LT, EQ, GT, or UN).

Each time a fragment program is executed, the fragment attribute registers are initialized with the fragment's location and associated data, all temporary register components are initialized to zero, and all condition code components are initialized to EQ.

Renumber Section 3.11 to Section 3.12, Antialiasing Application (p.140). No changes to the text of the section.

Additions to Chapter 4 of the OpenGL 1.2.1 Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the OpenGL 1.2.1 Specification (Special Functions)

Add new section 5.7, Programs (after "Flush and Finish")

Programs are specified as an array of ubytes used to control the operation of portions of the GL. The array is a string of ASCII characters encoding the program.

The command

```
  LoadProgramNV(enum target, uint id, sizei len, const ubyte *program);
```

loads a program. The target parameter specifies the type of program loaded and can be VERTEX_PROGRAM_NV, VERTEX_STATE_PROGRAM_NV, or FRAGMENT_PROGRAM_NV. VERTEX_PROGRAM_NV specifies a program to be executed in vertex program mode as each vertex is specified. VERTEX_STATE_PROGRAM specifies a program to be run manually to update vertex state. FRAGMENT_PROGRAM specifies a program to be executed in fragment program mode as each fragment is rasterized.

Multiple programs can be loaded with different names. id names the program to load. The name space for programs is the set of positive integers (zero is reserved). The error INVALID_VALUE is generated by LoadProgramNV if a program is loaded with an id of zero. The error INVALID_OPERATION is generated by LoadProgramNV or if a program is loaded for an id that is currently loaded with a program of a different program target. program is a pointer to an array of ubytes that represents the program being loaded. The length of the array in ubytes is indicated by len.

At program load time, the program is parsed into a set of tokens possibly separated by white space. Spaces, tabs, newlines, carriage returns, and comments are considered whitespace. Comments begin with the character "#" and are terminated by a newline, a carriage return, or the end of the program array. Tokens are processed in a case-sensitive manner: upper and lower-case letters are not considered equivalent.

Each program target has a corresponding Backus-Naur Form (BNF) grammar specifying the syntactically valid sequences for programs of the specified type. The set of valid tokens can be inferred from the grammar. The token "" represents an empty string and is used to indicate optional rules. A program is invalid if it contains any undefined tokens or characters.

The error INVALID_OPERATION is generated by LoadProgramNV if a program fails to load because it is not syntactically correct or fails to satisfy all of the semantic restrictions corresponding to the program target.

A successfully loaded program is parsed into a sequence of instructions. Each instruction is identified by its tokenized name. The operation of these instructions is specific to the program target and is defined elsewhere.

A successfully loaded program replaces the program previously assigned to the name specified by id. If the OUT_OF_MEMORY error is generated by LoadProgramNV, no change is made to the previous contents of the named program.

Querying the value of PROGRAM_ERROR_POSITION_NV returns a ubyte offset into the program string most recently passed to LoadProgramNV indicating the position of the first error, if any, in the program. If the program fails to load because of a semantic restriction that cannot be determined until the program is fully scanned, the error position will be len, the length of the program. If the program loads successfully, the value of PROGRAM_ERROR_POSITION_NV is assigned the value negative one.

For targets whose programs are executed automatically (e.g., vertex and fragment programs), there must be a current program. The current vertex program is executed automatically in vertex program mode as vertices are specified. The current fragment program is executed automatically in fragment program mode as fragments are generated by rasterization. Current programs for a program target are updated by

```
  BindProgramNV(enum target, uint id);
```

where target must be VERTEX_PROGRAM_NV or FRAGMENT_PROGRAM_NV. The error INVALID_OPERATION is generated by BindProgramNV if id names a program that has a type different than target (for example, if id names a vertex state program as described in section 2.14.4).

Binding to a nonexistent program id does not generate an error. In particular, binding to program id zero does not generate an error. However, because program zero cannot be loaded, program zero is always nonexistent. If a program id is successfully loaded with a new vertex program and id is also the currently bound vertex program, the new program is considered the currently bound vertex program.

The INVALID_OPERATION error is generated when both vertex program mode is enabled and Begin is called (or when a command that performs an implicit Begin is called) if the current vertex program is nonexistent or not valid. A vertex program may not be valid for reasons explained in section 2.14.5.

The INVALID_OPERATION error is generated when both fragment program mode is enabled and Begin, another GL command that performs an implicit Begin, or any other GL command that generates fragments is called, if the current fragment program is nonexistent or not valid. A fragment program may be invalid for reasons explained in Section 3.11.3.

Programs are deleted by calling

```
  void DeleteProgramsNV(sizei n, const uint *ids);
```

ids contains n names of programs to be deleted. After a program is deleted, it becomes nonexistent, and its name is again unused. If a program that is currently bound is deleted, it is as though BindProgramNV has been executed with the same target as the deleted program and program zero. Unused names in ids are silently ignored, as is the value zero.

The command

```
  void GenProgramsNV(sizei n, uint *ids);
```

returns n currently unused program names in ids. These names are marked as used, for the purposes of GenProgramsNV only, but they become existent programs only when the are first loaded using LoadProgramNV.

An implementation may choose to establish a working set of programs on which binding and/or manual execution are performed with higher performance. A program that is currently part of this working set is said to be resident.

The command

```
  boolean AreProgramsResidentNV(sizei n, const uint *ids,
                                boolean *residences);
```

returns TRUE if all of the n programs named in ids are resident, or if the implementation does not distinguish a working set. If at least one of the programs named in ids is not resident, then FALSE is returned, and the residence of each program is returned in residences. Otherwise the contents of residences are not changed. If any of the names in ids are nonexistent or zero, FALSE is returned, the error INVALID_VALUE is generated, and the contents of residences are indeterminate. The residence status of a single named program can also be queried by calling GetProgramivNV (Section 6.1.13) with id set to the name of the program and pname set to PROGRAM_RESIDENT_NV.

AreProgramsResidentNV indicates only whether a program is currently resident, not whether it could not be made resident. An implementation may choose to make a program resident only on first use, for example. The client may guide the GL implementation in determining which programs should be resident by requesting a set of programs to make resident.

The command

```
  void RequestResidentProgramsNV(sizei n, const uint *ids);
```

requests that the n programs named in ids should be made resident. While all the programs are not guaranteed to become resident, the implementation should make a best effort to make as many of the programs resident as possible. As a result of making the requested programs resident, program names not among the requested programs may become non-resident. Higher priority for residency should be given to programs listed earlier in the ids array. RequestResidentProgramsNV silently ignores attempts to make resident nonexistent program names or zero. AreProgramsResidentNV can be called after RequestResidentProgramsNV to determine which programs actually became resident.

The commands

```
  void ProgramNamedParameter4fNV(uint id, sizei len, const ubyte *name,
                                 float x, float y, float z, float w);
  void ProgramNamedParameter4dNV(uint id, sizei len, const ubyte *name,
                                 double x, double y, double z, double w);
  void ProgramNamedParameter4fvNV(uint id, sizei len, const ubyte *name,
                                  const float v[]);
  void ProgramNamedParameter4dvNV(uint id, sizei len, const ubyte *name,
                                  const double v[]);
```

specify a new value for the named program local parameter <name> belonging to the fragment program specified by <id>. <name> is a pointer to an array of ubytes holding the parameter name. <len> specifies the number of ubytes in the array given by <name>. The new x, y, z, and w components of the named local parameter are given by x, y, z, and w, respectively, for ProgramNamedParameter4fNV and ProgramNamedParameter4dNV, and by v[0], v[1], v[2], and v[3], respectively, for ProgramNamedParameter4fvNV and ProgramNamedParameter4dvNV. The error INVALID_OPERATION is generated if <id> specifies a nonexistent program or a program whose type does not suport named local parameters. The error INVALID_VALUE error is generated if <name> does not specify the name of a local parameter in the program corresponding to <id>. The error INVALID_VALUE is also generated if <len> is zero.

The commands

```
  void ProgramLocalParameter4fARB(enum target, uint index,
                                  float x, float y, float z, float w);
  void ProgramLocalParameter4fvARB(enum target, uint index,
                                   const float *params);
  void ProgramLocalParameter4dARB(enum target, uint index,
                                  double x, double y, double z, double w);
  void ProgramLocalParameter4dvARB(enum target, uint index,
                                   const double *params);
```

update the values of the numbered program local parameter <index> belonging to the program object currently bound to <target>. For ProgramLocalParameter4fARB and ProgramLocalParameter4dARB, the four components of the parameter are updated with the values of <x>, <y>, <z>, and <w>, respectively. For ProgramLocalParameter4fvARB and ProgramLocalParameter4dvARB, the four components of the parameter are updated with the array of four values pointed to by <params>. The error INVALID_VALUE is generated if <index> is greater than or equal to the number of numbered program local parameters supported by <target>.

Additions to Chapter 6 of the OpenGL 1.2.1 Specification (State and State Requests)

Modify Section 6.1.11, Pointer and String Queries (p. 206)

(modify last paragraph, p. 206) ... The possible values for <name> are VENDOR, RENDERER, VERSION, EXTENSIONS, and PROGRAM_ERROR_STRING_NV.

(add after last paragraph of section, p. 207) Queries of PROGRAM_ERROR_STRING_NV return a pointer to an implementation-dependent program load error string. If the last call to LoadProgramNV failed to load a program, the returned string describes a reason that the program failed to load. Otherwise, a pointer to an empty string (containing only a terminator) is returned.

Rename and modify Section 6.1.13, Vertex and Fragment Program Queries (from GL_NV_fragment_program). Portions of this section pertaining to fragment programs are copied verbatim.

(insert after discussion of GetProgramParameter[fd]vNV)

The commands

```
  void GetProgramNamedParameterfvNV(uint id, sizei len,
                                    const ubyte *name, float *params);
  void GetProgramNamedParameterdvNV(uint id, sizei len,
                                    const ubyte *name, double *params);
```

obtain the current program named local parameter value for the parameter named <name> belonging to the program given by <id>. <name> is a pointer to an array of ubytes holding the parameter name. <len> specifies the number of ubytes in the array given by <name>. The error INVALID_OPERATION is generated if <id> specifies a nonexistent program or a program whose type does not suport named local parameters. The error INVALID_VALUE is generated if <name> does not specify the name of a local parameter in the program corresponding to <id>. The error INVALID_VALUE is also generated if <len> is zero. Each named program local parameter is an array of four values.

The commands

```
  void GetProgramLocalParameterdvARB(enum target, uint index,
                                     double *params);
  void GetProgramLocalParameterfvARB(enum target, uint index,
                                     float *params);
```

obtain the current value for the numbered program local parameter <index> belonging to the program object currently bound to <target>, and places the information in the array <params>. The error INVALID_ENUM is generated if <target> specifies a nonexistent program target or a program target that does not support numbered program local parameters. The error INVALID_VALUE is generated if <index> is greater than or equal to the implementation-dependent number of supported numbered program local parameters for the program target.

When the program target type is FRAGMENT_PROGRAM_NV, each numbered program local parameter returned is an array of four values. ...

The command

```
  void GetProgramivNV(uint id, enum pname, int *params);
```

obtains program state named by pname for the program named id in the array params. pname must be one of PROGRAM_TARGET_NV, PROGRAM_LENGTH_NV, or PROGRAM_RESIDENT_NV. The error INVALID_OPERATION is generated if the program named id does not exist.

The command

```
  void GetProgramStringNV(uint id, enum pname,
                          ubyte *program);
```

obtains the program string for program id. pname must be PROGRAM_STRING_NV. n ubytes are returned into the array program where n is the length of the program in ubytes. GetProgramivNV with PROGRAM_LENGTH_NV can be used to query the length of a program's string. The INVALID_OPERATION error is generated if the program named id does not exist.

...

The command

```
  boolean IsProgramNV(uint id);
```

returns TRUE if program is the name of a program object. If program is zero or is a non-zero value that is not the name of a program object, or if an error condition occurs, IsProgramNV returns FALSE. A name returned by GenProgramsNV but not yet loaded with a program is not the name of a program object."

Additions to Appendix F of the OpenGL 1.2.1 Specification (ARB Extensions)

Modify Section F.2.3 (Changes to Section 2.6), p.240

(modify last paragraph on p.240) ... Multiple sets of texture coordinates may be used to specify how multiple texture images are mapped onto a primitive. The number of texture coordinate sets supported is implementation dependent, but must be at least 1. The number of texture coordinate sets supported may be queried with the state MAX_TEXTURE_COORDS_NV.

Modify Section F.2.4 (Changes to Section 2.7), p.241

(modify the last paragraph on p.241, carrying over to p.243) Implementations may support more than one set of texture coordinates. The commands

```
    void MultiTexCoord{1234}{sifd}ARB(enum texture, T coords)
    void MultiTexCoord{1234}{sifd}vARB(enum texture, T coords)
```

take the coordinate set to be modified as the <texture> parameter. <texture> is a symbolic constant of the form TEXTUREi_ARB, indicating that texture coordinate set i is to be modified. The constants obey TEXTUREi_ARB = TEXTURE0_ARB + i (i is in the range 0 to k-1, where k is the implementation dependent number of texture units defined by MAX_TEXTURE_COORDS_NV).

Modify Section F.2.5 (Changes to Section 2.8), p.243

(modify first and second paragraphs of section) ... The client may specify up to 5 plus the value of MAX_TEXTURE_COORDS_NV arrays; one each to store vertex coordinates...

In implementations which support more than one texture coordinate set, the command

```
    void ClientActiveTextureARB(enum texture)
```

is used to select the vertex array client state parameters to be modified by the TexCoordPointer command and the array affected by EnableClientState and DisableClientState with the parameter TEXTURE_COORD_ARRAY. This command sets the state variable CLIENT_ACTIVE_TEXTURE_ARB. Each texture coordinate set has a client state vector which is selected when this command is invoked. This state vector also includes the vertex array state. This command also selects the texture coordinate set state used for queries of client state.

(modify first paragraph on p.244) If the number of supported texture coordinate sets (the value of MAX_TEXTURE_COORDS_NV) is k, ...

Modify Section F.2.6 (Changes to Section 2.10.2), p.244

(modify first paragraph) For each texture coordinate set, a 4x4 matrix is applied to the corresponding texture coordinates...

(replace second and third paragraphs) The command

```
  void ActiveTextureARB(enum texture);
```

specifies the active texture unit selector, ACTIVE_TEXTURE_ARB. Each texture unit contains up to two distinct sub-units: a texture coordinate processing unit (consisting of a texture matrix stack and texture coordinate generation state) and a texture image unit (consisting of all the texture state defined in Section 3.8). In implementations with a different number of supported texture coordinate sets and texture image units, some texture units may consist of only one of the two sub-units.

The active texture unit selector specifies the texture unit accessed by commands involving texture coordinate processing. Such commands include those accessing the current matrix stack (if MATRIX_MODE is TEXTURE), TexGen (Section 2.10.4), Enable/Disable (if any texture coordinate generation enum is selected), as well as queries of the current texture coordinates and current raster texture coordinates. If the texture unit number corresponding to the current value of ACTIVE_TEXTURE_ARB is greater than or equal to the implementation dependent constant MAX_TEXTURE_COORD_SETS_NV, the error INVALID_OPERATION is generated by any such command.

The active texture unit selector also selects the texture unit accessed by commands involving texture image processing (Section 3.8). Such commands include all variants of TexEnv, TexParameter, and TexImage commands, BindTexture, Enable/Disable for any texture target (e.g., TEXTURE_2D), and queries of all such state. If the texture unit number corresponding to the current value of ACTIVE_TEXTURE_ARB is greater than or equal to the implementation dependent constant MAX_TEXTURE_IMAGE_UNITS_NV, the error INVALID_OPERATION is generated by any such command.

ActiveTextureARB generates the error INVALID_ENUM if an invalid <texture> is specified. <texture> is a symbolic constant of the form TEXTUREi_ARB, indicating that texture unit i is to be modified. The constants obey

TEXTUREi_ARB = TEXTURE0_ARB + i (i is in the range 0 to k-1, where k is the larger of the MAX_TEXTURE_COORDS_NV and MAX_TEXTURE_IMAGE_UNITS_NV). For compatibility with old OpenGL specifications, the implementation dependent constant MAX_TEXTURE_UNITS_ARB specifies the number of conventional texture units supported by the implementation. Its value must be no larger than the minimum of MAX_TEXTURE_COORDS_NV and MAX_TEXTURE_IMAGE_UNITS_NV.

Modify Section F.2.12 (Changes to Section 3.8.10), p.249

(modify next-to-last paragraph) Texturing is enabled and disabled individually for each texture unit. If texturing is disabled for one of the units, then the fragment resulting from the previous unit is passed unaltered to the following unit. Individual texture units beyond those specified by MAX_TEXTURE_UNITS_ARB may be incomplete and are always treated as disabled.

Modify Section F.2.15 (Changes to Section 6.1.2), p.251

(add to end of paragraph) Queries of texture state variables corresponding to texture coordinate processing unit (namely, TexGen state and enables, and matrices) will produce an INVALID_OPERATION error if the value of ACTIVE_TEXTURE_ARB is greater than or equal to MAX_TEXTURE_COORDS_NV. All other texture state queries will result in an INVALID_OPERATION error if the value of ACTIVE_TEXTURE_ARB is greater than or equal to MAX_TEXTURE_IMAGE_UNITS_NV.

Additions to the AGL/GLX/WGL Specifications

Program objects are shared between AGL/GLX/WGL rendering contexts if and only if the rendering contexts share display lists. No change is made to the AGL/GLX/WGL API.

Dependencies on GL_NV_vertex_program

If NV_vertex_program is supported, the description of LoadProgramNV in Section 2.14.1.7 (up to the BNF description of vertex programs) is deleted, as it is replaced by the contents of Section 5.7 in this specification. The general error descriptions in Section 2.14.1.7 common to Section 5.7 (like INVALID_OPERATION if the program fails to compile) should also be deleted. Section 2.14.1.8 should also be deleted. Section 6.1.13 is modified by this specification as described above.

Dependencies on NV_register_combiners

If NV_register_combiners is not supported, combiner programs are not supported, the TEX0, TEX1, TEX2, and TEX3 output registers are eliminated, and all references to both in this extension are deleted.

Dependencies on NV_texture_shader

If NV_texture_shader is not supported, the comment about texture shaders being disabled in fragment program mode is not applicable.

Dependencies on NV_texture_rectangle

    If NV_texture_rectangle is not supported, the references to "RECT" in the
    <texImageTarget> grammar rule and TEXTURE_RECTANGLE_NV are not applicable.

Dependencies on ARB_texture_cube_map

    If NV_texture_rectangle is not supported, the references to "CUBE" in the
    <texImageTarget> grammar rule and TEXTURE_CUBE_MAP_ARB are not applicable.

Dependencies on EXT_fog_coord

    If EXT_fog_coord is not supported, references to "fog coordinate" in the
    definition of the "FOGC" fragment attribute register should be removed.

Dependencies on NV_depth_clamp

    If NV_depth_clamp is not supported, section 3.11.6 is modified to remove
    discussion of the depth clamp enable and instead indicate that fragments
    with depth values outside [min(n,f), max(n,f)] are always discarded.

Dependencies on ARB_depth_texture and SGIX_depth_texture

    If ARB_depth_texture is not supported, but SGIX_depth_texture is
    supported, the discussion of Table X.5 is modified to indicate that
    DEPTH_COMPONENT textures are treated as LUMINANCE.

    If neither extension is supported, the discussion of DEPTH_COMPONENT
    textures in Table X.5 should be removed.

Dependencies on NV_float_buffer

    If NV_float_buffer is not supported, references to FLOAT_R_NV,
    FLOAT_RG_NV, FLOAT_RGB_NV, and FLOAT_RGBA_NV internal texture formats in
    Table X.5 should be removed.

Dependencies on ARB_vertex_program

    This extension does not have any explicit dependencies, but the APIs for
    setting and querying numbered local parameters (ProgramLocalParameter*ARB
    and GetProgramLocalParameter*ARB) were taken directly from this extension,

GLX Protocol

    Most of the GLX protocol needed to implement this extension is described
    in the GL_NV_vertex_program extension specification and will not be
    repeated here.

```
    The following two rendering commands are potentially large, and hence can
    be sent in a glXRender or glXRenderLarge request.

        ProgramNamedParameter4fvNV
            2           28+len+p        rendering command length
            2           4218            rendering command opcode
            4           CARD32          id
            4           CARD32          len
            4           FLOAT32         params[0]
            4           FLOAT32         params[1]
            4           FLOAT32         params[2]
            4           FLOAT32         params[3]
            len         LISTofCARD8     name
            p                           unused, p=pad(len)

         If the command is encoded in a glxRenderLarge request, the command
         opcode and command length fields above are expanded to 4 bytes each:

            4           32+len+p        rendering command length
            4           4218            rendering command opcode

        ProgramNamedParameter4dvNV
            2           44+len+p        rendering command length
            2           4219            rendering command opcode
            4           CARD32          id
            4           CARD32          len
            8           FLOAT64         params[0]
            8           FLOAT64         params[1]
            8           FLOAT64         params[2]
            8           FLOAT64         params[3]
            len         LISTofCARD8     name
            p                           unused, p=pad(len)

         If the command is encoded in a glxRenderLarge request, the command
         opcode and command length fields above are expanded to 4 bytes each:

            4           48+len+p        rendering command length
            4           4219            rendering command opcode
    The remaining two commands are non-rendering commands.  These commands are
    sent separately (i.e., not as part of a glXRender or glXRenderLarge
    request), using the glXVendorPrivateWithReply request:

        GetProgramNamedParameter4fvNV
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           4+(len+p)/4     request length
            4           1310            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           INT32           len
            len         LISTofCARD8     name
            p                           unused, p=pad(len)
          =>

          If the command succeeds, 4 floats are sent in the reply:

            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           4               reply length
            24                          unused
            16          LISTofFLOAT32   params

          Otherwise, an empty reply is sent, indicating that a GL error
          occured:

            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           0               reply length
            24                          unused
        GetProgramNamedParameter4dvNV
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           4+(len+p)/4     request length
            4           1311            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           INT32           len
            len         LISTofCARD8     name
            p                           unused, p=pad(len)
          =>

          If the command succeeds, 4 doubles are sent in the reply:

            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           8               reply length
            24                          unused
            32          LISTofFLOAT64   params

          Otherwise, an empty reply is sent, indicating that a GL error
          occured:

            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           0               reply length
            24                          unused
```

Errors

INVALID_OPERATION is generated by Begin, DrawPixels, Bitmap, CopyPixels, or a command that performs an explicit Begin if FRAGMENT_PROGRAM_NV is enabled and the currently bound fragment program does not exist.

INVALID_OPERATION is generated by ProgramNamedParameter4fNV, ProgramNamedParameter4dNV, ProgramNamedParameter4fvNV, ProgramNamedParameter4dvNV, GetProgramNamedParameterfvNV, or GetProgramNamedParameterdvNV if <id> specifies a nonexistent program or a program whose type does not suport local parameters.

INVALID_VALUE is generated by ProgramNamedParameter4fNV, ProgramNamedParameter4dNV, ProgramNamedParameter4fvNV, ProgramNamedParameter4dvNV, GetProgramNamedParameterfvNV, or GetProgramNamedParameterdvNV if <len> is zero.

INVALID_VALUE is generated by ProgramNamedParameter4fNV, ProgramNamedParameter4dNV, ProgramNamedParameter4fvNV, ProgramNamedParameter4dvNV, GetProgramNamedParameterfvNV, or GetProgramNamedParameterdvNV if <name> does not specify the name of a local parameter in the program corresponding to <id>.

INVALID_OPERATION is generated by any command accessing texture coordinate processing state if the texture unit number corresponding to the current value of ACTIVE_TEXTURE_ARB is greater than or equal to the implementation dependent constant MAX_TEXTURE_COORD_SETS_NV.

INVALID_OPERATION is generated by any command accessing texture image processing state if the texture unit number corresponding to the current value of ACTIVE_TEXTURE_ARB is greater than or equal to the implementation dependent constant MAX_TEXTURE_IMAGE_UNITS_NV.

(The following are error descriptions copied from GL_NV_vertex_program that apply to this extension as well. These modifications do not affect the behavior of that extension.)

INVALID_VALUE is generated by LoadProgramNV if id is zero.

INVALID_OPERATION is generated by LoadProgramNV if the program corresponding to id is currently loaded but has a program type different from that given by target.

INVALID_OPERATION is generated by LoadProgramNV if the program specified is syntactically incorrect for the program type specified by target. The value of PROGRAM_ERROR_POSITION_NV is still updated when this error is generated.

INVALID_OPERATION is generated by LoadProgramNV if the problem specified fails to conform to any of the semantic restrictions imposed on programs of the type specified by target. The value of PROGRAM_ERROR_POSITION_NV is still updated when this error is generated.

INVALID_OPERATION is generated by BindProgramNV if target does not match the type of the program named by id.

INVALID_VALUE is generated by AreProgramsResidentNV if any of the queried programs are zero or do not exist.

INVALID_OPERATION is generated by GetProgramivNV or GetProgramStringNV if the program named id does not exist.

New State

Get Value	Type	Get Command	Initial Value	Description	Section	Attribute
FRAGMENT_PROGRAM_NV	B	IsEnabled	FALSE	fragment program mode enable	3.11	enable
FRAGMENT_PROGRAM_BINDING_NV	Z+	GetIntegerv	0	bound fragment program	5.7	-

Table X.6. New State Introduced by NV_fragment_program.

Get Value	Type	Get Command	Initial Value	Description	Section	Attribute
PROGRAM_ERROR_POSITION_NV	Z	GetIntegerv	-1	program error position	5.7	-
PROGRAM_TARGET_NV	Z2	GetProgramivNV	0	program target	6.1.13	-
PROGRAM_LENGTH_NV	Z+	GetProgramivNV	0	program length	6.1.13	-
PROGRAM_RESIDENT_NV	Z2	GetProgramivNV	False	program residency	6.1.13	-
PROGRAM_STRING_NV	ubxn	GetProgramStringNV	""	program string	6.1.13	-
-	nxR4	GetProgramNamed-ParameterNV	(0,0,0,0)	named program local parameter value	5.7	-
-	64+xR4	GetProgramLocal-ParameterARB	(0,0,0,0)	numbered program local parameter	5.7	-

Table X.7. Program Object State common to NV_vertex_program and NV_fragment_program.

Get Value	Type	Get Command	Initial Value	Description	Section	Attribute
-	12xR4	-	fragment data	fragment attribute registers	3.11.1.1	-
-	16xR4	-	(0,0,0,0)	fp32 temporary registers	3.11.1.2	-
-	32xR4	-	(0,0,0,0)	fp16 temporary registers	3.11.1.2	-
	(Z_4)4	-	(EQ,EQ,EQ,EQ)	condition code register address register	3.11.1.4	-

Table X.8. Fragment Program Per-Fragment Execution State.

New Implementation Dependent State

Get Value	Type	Get Command	Minimum Value	Description	Section	Attribute
MAX_TEXTURE_COORDS_NV	Z+	GetIntegerv	2	number of texture coordinate sets supported	2.6	-
MAX_TEXTURE_IMAGE_UNITS_NV	Z+	GetIntegerv	2	number of texture image units supported	2.10.2	-
MAX_FRAGMENT_PROGRAM_ LOCAL_PARAMETERS_NV	Z+	GetIntegerv	64	number of numbered local parameters supported	3.11.7	-

Name

    NV_fragment_program_option

Name Strings

    GL_NV_fragment_program_option

Status

    Shipping.

Version

```
Last Modified:      05/27/2005
NVIDIA Revision:    4
```

Number

    303

Dependencies

    ARB_fragment_program is required.

Overview

This extension provides additional fragment program functionality to extend the standard ARB_fragment_program language and execution environment. ARB programs wishing to use this added functionality need only add:

```
OPTION NV_fragment_program;
```

to the beginning of their fragment programs.

The functionality provided by this extension, which is roughly equivalent to that provided by the NV_fragment_program extension, includes:

* increased control over precision in arithmetic computations and storage,

* data-dependent conditional writemasks,

* an absolute value operator on scalar and swizzled operand loads,

* instructions to compute partial derivatives, and perform texture lookups using specified partial derivatives,

* fully orthogonal "set on" instructions,

* instructions to compute reflection vector and perform a 2D coordinate transform, and

* instructions to pack and unpack multiple quantities into a single component.

Issues

Why is this a separate extension, rather than just an additional feature of NV_fragment_program?

RESOLVED: The NV_fragment_program specification was complete (with a published implementation) prior to the completion of ARB_fragment_program. Future NVIDIA fragment program extensions should contain extensions to the ARB_fragment_program execution environment as a standard feature.

Should a similar option be provided to expose ARB_fragment_program features not found in NV_fragment_program (e.g., state bindings, certain "macro" instructions) under the NV_fragment_program interface?

RESOLVED: No. Why not just write an ARB program?

The ARB_fragment_program spec has a minor grammar bug that requires that inline scalar constants used as scalar operands include a component selector. In other words, you have to say "11.0.x" to use the constant "11.0". What should we do here?

RESOLVED: The NV_fragment_program_option grammar will correct this problem, which should be fixed in future revisions to the ARB language.

New Procedures and Functions

None.

New Tokens

None.

Additions to Chapter 2 of the OpenGL 1.2.1 Specification (OpenGL Operation)

None.

Additions to Chapter 3 of the OpenGL 1.2.1 Specification (Rasterization)

Modify Section 3.11.2 of ARB_fragment_program (Fragment Program Grammar and Restrictions):

(mostly add to existing grammar rules, modify a few existing grammar rules -- changes marked with "***")

```
<optionName>            ::= "NV_fragment_program"

<TexInstruction>        ::= <TXDop_instruction>
```

```
    <VECTORop>              ::= "DDX"
                              | "DDY"
                              | "PK2H"
                              | "PK2US"
                              | "PK4B"
                              | "PK4UB"

    <SCALARop>              ::= "UP2H"
                              | "UP2US"
                              | "UP4B"
                              | "UP4UB"

    <BINop>                 ::= "RFL"
                              | "SEQ"
                              | "SFL"
                              | "SGT"
                              | "SLE"
                              | "SNE"
                              | "STR"

    <TRIop>                 ::= "X2D"

    <TXDop_instruction>     ::= <TXDop> <instResult> "," <instOperandV> ","
                                <instOperandV> "," <instOperandV> ","
                                <texTarget>

    <TXDop>                 ::= "TXD"

    <killCond>              ::= <ccTest>

    <instOperandV>          ::= <instOperandAbsV>

    <instOperandAbsV>       ::= <optSign> "|" <instOperandBaseV> "|"

    <instOperandS>          ::= <instOperandAbsS>

    <instOperandAbsS>       ::= <optSign> "|" <instOperandBaseS> "|"

    <instResult>            ::= <instResultCC>

    <instResultCC>          ::= <instResultBase> <ccMask>

    <TEMP_statement>        ::= <varSize> "TEMP" <varNameList>

    <OUTPUT_statement>      ::= <varSize> "OUTPUT" <establishName> "="
                                  <resultUseD>

    <varSize>               ::= "SHORT"
                              | "LONG"

    <paramUseV>             ::= <constantScalar>
                                  (*** instead of <constantScalar>
                                       <swizzleSuffix>)

    <paramUseS>             ::= <constantScalar>
                                  (*** instead of <constantScalar>
                                       <scalarSuffix>)
```

```
<ccMask>                ::= "(" <ccTest> ")"

<ccTest>                ::= <ccMaskRule> <swizzleSuffix>

<ccMaskRule>            ::= "EQ"
                          | "GE"
                          | "GT"
                          | "LE"
                          | "LT"
                          | "NE"
                          | "TR"
                          | "FL"
```

(modify language describing reserved keywords) The following strings are reserved keywords and may not be used as identifiers:

    ALIAS, ATTRIB, END, OPTION, OUTPUT, PARAM, TEMP, fragment,
    program, result, state, and texture.

Additionally, all the instruction names (and variants) listed in Table X.5 are reserved.

Modify Section 3.11.3.3, Fragment Program Temporaries

(replace second paragraph) Fragment program temporary variables can be declared explicitly using the <TEMP_statement> grammar rule. Each such statement can declare one or more temporaries. Temporary declaration can optionally specify a variable size, using the <varSize> grammar rule. Variables declared as "SHORT" will represented with at least 16 bits per component (5 bits of exponent, 10 bits of mantissa). Variables declared as "LONG" will be represented with at least 32 bits per component (8 bits of exponent, 23 bits of mantissa). Fragment program temporary variables can not be declared implicitly.

Modify Section 3.11.3.4, Fragment Program Results

(replace second paragraph) Fragment program result variables can be declared explicitly using the <OUTPUT_statement> grammar rule, or implicitly using the <resultBinding> grammar rule in an executable instruction. Explicit result variable declaration can optionally specify a variable size, using the <varSize> grammar rule. Variables declared as "SHORT" will represented with at least 16 bits per component (5 bits of exponent, 10 bits of mantissa). Variables declared as "LONG" will be represented with at least 32 bits per component (8 bits of exponent, 23 bits of mantissa). Each fragment program result variable is bound to a fragment attribute used in subsequent back-end processing. The set of fragment program result variable bindings is given in Table X.3.

(add to the end of a section) A fragment program will fail to load if contains instructions writing to variables bound to the same result, but declared with different sizes.

Add New Section 3.11.3.X, Condition Code Register (insert after Section 3.11.3.4, Fragment Program Results)

The fragment program condition code register is a single four-component vector. Each component of this register is one of four enumerated values: GT (greater than), EQ (equal), LT (less than), or UN (unordered). The condition code register can be used to mask writes to registers and to evaluate conditional branches.

Most fragment program instructions can optionally update the condition code register. When a fragment program instruction updates the condition code register, a condition code component is set to LT if the corresponding component of the result is less than zero, EQ if it is equal to zero, GT if it is greater than zero, and UN if it is NaN (not a number).

The condition code register is initialized to a vector of EQ values each time a fragment program executes.

Modify Section 3.11.4, Fragment Program Execution Environment

(modify instruction table) There are fifty-two fragment program instructions. Fragment program instructions may have up to sixteen variants, including a suffix of "R", "H", or "X" to specify arithmetic precision (section 3.11.4.X), a suffix of "C" to allow an update of the condition code register (section 3.11.3.X), and a suffix of "_SAT" to clamp the result vector components to the range [0,1] (section 3.11.4.3). For example, the sixteen forms of the "ADD" instruction are "ADD", "ADDR", "ADDH", "ADDX", "ADDC", "ADDRC", "ADDHC", "ADDXC", "ADD_SAT", "ADDR_SAT", "ADDH_SAT", "ADDX_SAT", "ADDC_SAT", "ADDRC_SAT", "ADDHC_SAT", and "ADDXC_SAT".The instructions and their respective input and output parameters are summarized in Table X.5.

	Modifiers							
Instr.	**R**	**H**	**X**	**C**	**S**	**Inputs**	**Output**	**Description**
ABS	X	X	X	X	X	v	v	absolute value
ADD	X	X	X	X	X	v,v	v	add
CMP	-	-	-	-	X	v,v,v	v	compare
COS	X	X	-	X	X	s	ssss	cosine with reduction to [-PI,PI]
DDX	X	X	-	X	X	v	v	partial derivative relative to X
DDY	X	X	-	X	X	v	v	partial derivative relative to Y
DP3	X	X	X	X	X	v,v	ssss	3-component dot product
DP4	X	X	X	X	X	v,v	ssss	4-component dot product
DPH	X	X	X	X	X	v,v	ssss	homogeneous dot product
DST	X	X	-	X	X	v,v	v	distance vector
EX2	X	X	-	X	X	s	ssss	exponential base 2
FLR	X	X	X	X	X	v	v	floor
FRC	X	X	X	X	X	v	v	fraction
KIL	-	-	-	-	-	v or c	-	kill fragment
LG2	X	X	-	X	X	s	ssss	logarithm base 2
LIT	X	X	-	X	X	v	v	compute light coefficients
LRP	X	X	X	X	X	v,v,v	v	linear interpolation
MAD	X	X	X	X	X	v,v,v	v	multiply and add
MAX	X	X	X	X	X	v,v	v	maximum
MIN	X	X	X	X	X	v,v	v	minimum

	Modifiers							
Instr.	**R**	**H**	**X**	**C**	**S**	**Inputs**	**Output**	**Description**
MOV	X	X	X	X	X	v	v	move
MUL	X	X	X	X	X	v,v	v	multiply
PK2H	-	-	-	-	-	v	ssss	pack two 16-bit floats
PK2US	-	-	-	-	-	v	ssss	pack two unsigned 16-bit scalars
PK4B	-	-	-	-	-	v	ssss	pack four signed 8-bit scalars
PK4UB	-	-	-	-	-	v	ssss	pack four unsigned 8-bit scalars
POW	X	X	-	X	X	s,s	ssss	exponentiate
RCP	X	X	-	X	X	s	ssss	reciprocal
RFL	X	X	-	X	X	v,v	v	reflection vector
RSQ	X	X	-	X	X	s	ssss	reciprocal square root
SCS	-	-	-	-	X	s	ss--	sine/cosine without reduction
SEQ	X	X	X	X	X	v,v	v	set on equal
SFL	X	X	X	X	X	v,v	v	set on false
SGE	X	X	X	X	X	v,v	v	set on greater than or equal
SGT	X	X	X	X	X	v,v	v	set on greater than
SIN	X	X	-	X	X	s	ssss	sine with reduction to [-PI,PI]
SLE	X	X	X	X	X	v,v	v	set on less than or equal
SLT	X	X	X	X	X	v,v	v	set on less than
SNE	X	X	X	X	X	v,v	v	set on not equal
STR	X	X	X	X	X	v,v	v	set on true
SUB	X	X	X	X	X	v,v	v	subtract
SWZ	-	-	-	-	X	v	v	extended swizzle
TEX	-	-	-	X	X	v	v	texture sample
TXB	-	-	-	X	X	v	v	texture sample with bias
TXD	-	-	-	X	X	v,v,v	v	texture sample w/partials
TXP	-	-	-	X	X	v	v	texture sample with projection
UP2H	-	-	-	X	X	s	v	unpack two 16-bit floats
UP2US	-	-	-	X	X	s	v	unpack two unsigned 16-bit scalars
UP4B	-	-	-	X	X	s	v	unpack four signed 8-bit scalars
UP4UB	-	-	-	X	X	s	v	unpack four unsigned 8-bit scalars
X2D	X	X	-	X	X	v,v,v	v	2D coordinate transformation
XPD	-	-	-	-	X	v,v	v	cross product

Table X.5: Summary of fragment program instructions. The columns "R", "H", "X", "C", and "S" indicate whether the "R", "H", or "X" precision modifiers, the C condition code update modifier, and the "_SAT" saturation modifier, respectively, are supported for the opcode. In the input/output columns, "v" indicates a floating-point vector input or output, "s" indicates a floating-point scalar input, "ssss" indicates a scalar output replicated across a 4-component result vector, "ss--" indicates two scalar outputs in the first two components, and "c" indicates a condition code test. Instructions describe as "texture sample" also specify a texture image unit identifier and a texture target.

Modify Section 3.11.4.1, Fragment Program Operands

(add prior to the discussion of negation) A component-wise absolute value operation can optionally performed on the operand if the operand is surrounded with two "|" characters. For example, "|src|" indicates that a component-wise absolute value operation should be performed on the variable named "src". In terms of the grammar, this operation is performed if the <instOperandV> or <instOperandS> grammar rules match <instOperandAbsV> or <instOperandAbsS>, respectively.

(modify operand load pseudo-code) The following pseudo-code spells out the operand generation process. In the example, "float" is a floating-point scalar type, while "floatVec" is a four-component vector. "source" refers to the register used for the operand, matching the <srcReg> rule. "abs" is TRUE if an absolute value operation should be performed on the operand (<instOperandAbsV> or <instOperandAbsS> rules) "negate" is TRUE if the <optionalSign> rule in <scalarSrcReg> or <swizzleSrcReg> matches "-" and FALSE otherwise. The ".c***", ".*c**", ".**c*", ".***c" modifiers refer to the x, y, z, and w components obtained by the swizzle operation; the ".c" modifier refers to the single component selected for a scalar load.

```
  floatVec VectorLoad(floatVec source)
  {
      floatVec operand;

      operand.x = source.c***;
      operand.y = source.*c**;
      operand.z = source.**c*;
      operand.w = source.***c;
      if (abs) {
         operand.x = abs(operand.x);
         operand.y = abs(operand.y);
         operand.z = abs(operand.z);
         operand.w = abs(operand.w);
      }
      if (negate) {
         operand.x = -operand.x;
         operand.y = -operand.y;
         operand.z = -operand.z;
         operand.w = -operand.w;
      }

      return operand;
  }

  float ScalarLoad(floatVec source)
  {
      float operand;

      operand = source.c;
      if (abs) {
        operand = abs(operand);
      if (negate) {
        operand = -operand;
      }

      return operand;
  }
```

Add New Section 3.11.4.X, Fragment Program Operation Precision (insert after Section 3.11.4,2, Fragment Program Parameter Arrays)

Fragment program implementations may be able to perform instructions with different levels of arithmetic precision. The "R", "H", and "X" opcode precision modifiers (Section 3.11.4) specify the minimum precision used to perform arithmetic operations. Instructions with an "R" precision modifiers will be carried out at no less than IEEE single-precision floating-point (8 bits of exponent, 23 bits of mantissa). Instructions with an "H" precision modifier will be carried out at no less than 16-bit floating-point precision (5 bits of exponent, 10 bits of mantissa). Instructions with an "X" precision modifier will be carried out at no less than signed 12-bit fixed-point precision (two's complement with 10 fraction bits).

If the result of a computation overflows the range of numbers supported by the instruction precision, the result will be +/-INF (infinity) for "R" and "H" precision, or -2048/1024 or +2047/1024 for "X" precision.

If no precision modifier is specified, the instruction will be carried out with at least as much precision as the destination variable.

Rewrite Section 3.11.4.3, Fragment Program Destination Register Update

Most fragment program instructions write a 4-component result vector to a single temporary or fragment result register. Writes to individual components of the destination register are controlled by individual component write masks specified as part of the instruction.

The component write mask is specified by the <optionalMask> rule found in the <maskedDstReg> rule. If the optional mask is "", all components are enabled. Otherwise, the optional mask names the individual components to enable. The characters "x", "y", "z", and "w" match the x, y, z, and w components, respectively. For example, an optional mask of ".xzw" indicates that the x, z, and w components should be enabled for writing but the y component should not. The grammar requires that the destination register mask components must be listed in "xyzw" order.

The condition code write mask is specified by the <ccMask> rule found in the <instResultCC> rule. The condition code register is loaded and swizzled according to the swizzle codes specified by <swizzleSuffix>. Each component of the swizzled condition code is tested according to the rule given by <ccMaskRule>. <ccMaskRule> may have the values "EQ", "NE", "LT", "GE", LE", or "GT", which mean to enable writes if the corresponding condition code field evaluates to equal, not equal, less than, greater than or equal, less than or equal, or greater than, respectively. Comparisons involving condition codes of "UN" (unordered) evaluate to true for "NE" and false otherwise. For example, if the condition code is (GT,LT,EQ,GT) and the condition code mask is "(NE.zyxw)", the swizzle operation will load (EQ,LT,GT,GT) and the mask will thus will enable writes on the y, z, and w components. In addition, "TR" always enables writes and "FL" always disables writes, regardless of the condition code. If the condition code mask is empty, it is treated as "(TR)".

Each component of the destination register is updated with the result of the fragment program instruction if and only if the component is enabled for writes by both the component write mask and the condition code write mask. Otherwise, the component of the destination register remains unchanged.

A fragment program instruction can also optionally update the condition code register. The condition code is updated if the condition code register update suffix "C" is present in the instruction. The instruction "ADDC" will update the condition code; the otherwise equivalent instruction "ADD" will not. If condition code updates are enabled, each component of the destination register enabled for writes is compared to zero. The corresponding component of the condition code is set to "LT", "EQ", or "GT", if the written component is less than, equal to, or greater than zero, respectively. Condition code components are set to "UN" if the written component is NaN (not a number). Values of -0.0 and +0.0 both evaluate to "EQ". If a component of the destination register is not enabled for writes, the corresponding condition code component is also unchanged.

In the following example code,

```
    # R1=(-2, 0, 2, NaN)              R0                 CC
    MOVC R0, R1;               # ( -2,  0,   2, NaN) (LT,EQ,GT,UN)
    MOVC R0.xyz, R1.yzwx;      # (  0,  2, NaN, NaN) (EQ,GT,UN,UN)
    MOVC R0 (NE), R1.zywx;     # (  0,  0, NaN,  -2) (EQ,EQ,UN,LT)
```

the first instruction writes (-2,0,2,NaN) to R0 and updates the condition code to (LT,EQ,GT,UN). The second instruction, only the "x", "y", and "z" components of R0 and the condition code are updated, so R0 ends up with (0,2,NaN,NaN) and the condition code ends up with (EQ,GT,UN,UN). In the third instruction, the condition code mask disables writes to the x component (its condition code field is "EQ"), so R0 ends up with (0,0,NaN,-2) and the condition code ends up with (EQ,EQ,UN,LT).

The following pseudocode illustrates the process of writing a result vector to the destination register. In the pseudocode, "instrmask" refers to the component write mask given by the <optWriteMask> rule. "ccMaskRule" refers to the condition code mask rule given by <ccMask> and "updatecc" is TRUE if and only if condition code updates are enabled. "result", "destination", and "cc" refer to the result vector, the register selected by <dstRegister> and the condition code, respectively. Condition codes do not exist in the VP1 execution environment.

```
  boolean TestCC(CondCode field) {
      switch (ccMaskRule) {
      case "EQ":  return (field == "EQ");
      case "NE":  return (field != "EQ");
      case "LT":  return (field == "LT");
      case "GE":  return (field == "GT" || field == "EQ");
      case "LE":  return (field == "LT" || field == "EQ");
      case "GT":  return (field == "GT");
      case "TR":  return TRUE;
      case "FL":  return FALSE;
      case "":    return TRUE;
      }
  }
```

```
      enum GenerateCC(float value) {
        if (value == NaN) {
          return UN;
        } else if (value < 0) {
          return LT;
        } else if (value == 0) {
          return EQ;
        } else {
          return GT;
        }
      }

      void UpdateDestination(floatVec destination, floatVec result)
      {
          floatVec merged;
          ccVec    mergedCC;

          // Merge the converted result into the destination register, under
          // control of the compile- and run-time write masks.
          merged = destination;
          mergedCC = cc;
          if (instrMask.x && TestCC(cc.c***)) {
              merged.x = result.x;
              if (updatecc) mergedCC.x = GenerateCC(result.x);
          }
          if (instrMask.y && TestCC(cc.*c**)) {
              merged.y = result.y;
              if (updatecc) mergedCC.y = GenerateCC(result.y);
          }
          if (instrMask.z && TestCC(cc.**c*)) {
              merged.z = result.z;
              if (updatecc) mergedCC.z = GenerateCC(result.z);
          }
          if (instrMask.w && TestCC(cc.***c)) {
              merged.w = result.w;
              if (updatecc) mergedCC.w = GenerateCC(result.w);
          }

          // Write out the new destination register and condition code.
          destination = merged;
          cc = mergedCC;
      }
```

    Add to Section 3.11.4.5 of ARB_fragment_program (Fragment Program
    Options):

Section 3.11.4.5.3, NV_fragment_program Option

If a fragment program specifies the "NV_fragment_program" option, the grammar will be extended to support the features found in the NV_fragment_program extension not present in the ARB_fragment_program extension, including:

* the availability of the following instructions:

    - DDX (partial derivative relative to X),
    - DDY (partial derivative relative to Y),
    - PK2H (pack as two half floats),
    - PK2US (pack as two unsigned shorts),
    - PK4B (pack as four signed bytes),
    - PK4UB (pack as four unsigned bytes),
    - RFL (reflection vector),
    - SEQ (set on equal to),
    - SFL (set on false),
    - SGT (set on greater than),
    - SLE (set on less than or equal to),
    - SNE (set on not equal to),
    - STR (set on true),
    - TXD (texture lookup with computed partial derivatives),
    - UP2H (unpack two half floats),
    - UP2US (unpack two unsigned shorts),
    - UP4B (unpack four signed bytes),
    - UP4UB (unpack four unsigned bytes), and
    - X2D (2D coordinate transformation),

* opcode precision suffixes "R", "H", and "X", to specify the precision of arithmetic operations ("R" specifies 32-bit floating-point computations, "H" specifies 16-bit floating-point computations, and "X" specifies 12-bit signed fixed-point computations with 10 fraction bits),

* the availability of the "SHORT" and "LONG" variable precision keywords to control the size of a variable's components,

* a four-component condition code register to hold the sign of result vector components (useful for comparisons),

* a condition code update opcode suffix "C", where the results of the instruction are used to update the condition code register,

* a condition code write mask operator, where the condition code register is swizzled and tested, and the test results are used to mask register writes,

* an absolute value operator on scalar and swizzled source inputs

The added functionality is identical to that provided by the NV_fragment_program extension specification.

Modify Section 3.11.5, Fragment Program ALU Instruction Set

Section 3.11.5.30, DDX: Derivative Relative to X

The DDX instruction computes approximate partial derivatives of the four components of the single operand with respect to the X window coordinate to yield a result vector. The partial derivatives are evaluated at the center of the pixel.

```
  f = VectorLoad(op0);
  result = ComputePartialX(f);
```

Note that the partial derivates obtained by this instruction are approximate, and derivative-of-derivate instruction sequences may not yield accurate second derivatives.

Section 3.11.5.31, DDY: Derivative Relative to Y

The DDY instruction computes approximate partial derivatives of the four components of the single operand with respect to the Y window coordinate to yield a result vector. The partial derivatives are evaluated at the center of the pixel.

```
  f = VectorLoad(op0);
  result = ComputePartialY(f);
```

Note that the partial derivates obtained by this instruction are approximate, and derivative-of-derivate instruction sequences may not yield accurate second derivatives.

Section 3.11.5.32, PK2H: Pack Two 16-bit Floats

The PK2H instruction converts the "x" and "y" components of the single operand into 16-bit floating-point format, packs the bit representation of these two floats into a 32-bit value, and replicates that value to all four components of the result vector. The PK2H instruction can be reversed by the UP2H instruction below.

```
  tmp0 = VectorLoad(op0);
  /* result obtained by combining raw bits of tmp0.x, tmp0.y */
  result.x = RawBits(tmp0.x) | (RawBits(tmp0.y) << 16);
  result.y = RawBits(tmp0.x) | (RawBits(tmp0.y) << 16);
  result.z = RawBits(tmp0.x) | (RawBits(tmp0.y) << 16);
  result.w = RawBits(tmp0.x) | (RawBits(tmp0.y) << 16);
```

A fragment program will fail to load if it contains a PK2H instruction that writes its results to a variable declared as "SHORT".

Section 3.11.5.33, PK2US: Pack Two Unsigned 16-bit Scalars

The PK2US instruction converts the "x" and "y" components of the single operand into a packed pair of 16-bit unsigned scalars. The scalars are represented in a bit pattern where all '0' bits corresponds to 0.0 and all '1' bits corresponds to 1.0. The bit representations of the two converted components are packed into a 32-bit value, and that value is replicated to all four components of the result vector. The PK2US instruction can be reversed by the UP2US instruction below.

```
  tmp0 = VectorLoad(op0);
  if (tmp0.x < 0.0) tmp0.x = 0.0;
  if (tmp0.x > 1.0) tmp0.x = 1.0;
  if (tmp0.y < 0.0) tmp0.y = 0.0;
  if (tmp0.y > 1.0) tmp0.y = 1.0;
  us.x = round(65535.0 * tmp0.x);  /* us is a ushort vector */
  us.y = round(65535.0 * tmp0.y);
  /* result obtained by combining raw bits of us. */
  result.x = ((us.x) | (us.y << 16));
  result.y = ((us.x) | (us.y << 16));
  result.z = ((us.x) | (us.y << 16));
  result.w = ((us.x) | (us.y << 16));
```

A fragment program will fail to load if it contains a PK2S instruction that writes its results to a variable declared as "SHORT".

Section 3.11.5.34, PK4B: Pack Four Signed 8-bit Scalars

The PK4B instruction converts the four components of the single operand into 8-bit signed quantities. The signed quantities are represented in a bit pattern where all '0' bits corresponds to -128/127 and all '1' bits corresponds to +127/127. The bit representations of the four converted components are packed into a 32-bit value, and that value is replicated to all four components of the result vector. The PK4B instruction can be reversed by the UP4B instruction below.

```
  tmp0 = VectorLoad(op0);
  if (tmp0.x < -128/127) tmp0.x = -128/127;
  if (tmp0.y < -128/127) tmp0.y = -128/127;
  if (tmp0.z < -128/127) tmp0.z = -128/127;
  if (tmp0.w < -128/127) tmp0.w = -128/127;
  if (tmp0.x > +127/127) tmp0.x = +127/127;
  if (tmp0.y > +127/127) tmp0.y = +127/127;
  if (tmp0.z > +127/127) tmp0.z = +127/127;
  if (tmp0.w > +127/127) tmp0.w = +127/127;
  ub.x = round(127.0 * tmp0.x + 128.0);  /* ub is a ubyte vector */
  ub.y = round(127.0 * tmp0.y + 128.0);
  ub.z = round(127.0 * tmp0.z + 128.0);
  ub.w = round(127.0 * tmp0.w + 128.0);
  /* result obtained by combining raw bits of ub. */
  result.x = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
  result.y = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
  result.z = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
  result.w = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
```

A fragment program will fail to load if it contains a PK4B instruction that writes its results to a variable declared as "SHORT".

Section 3.11.5.35, PK4UB: Pack Four Unsigned 8-bit Scalars

The PK4UB instruction converts the four components of the single operand into a packed grouping of 8-bit unsigned scalars. The scalars are represented in a bit pattern where all '0' bits corresponds to

0.0 and all '1' bits corresponds to 1.0. The bit representations of the four converted components are packed into a 32-bit value, and that value is replicated to all four components of the result vector. The PK4UB instruction can be reversed by the UP4UB instruction below.

```
  tmp0 = VectorLoad(op0);
  if (tmp0.x < 0.0) tmp0.x = 0.0;
  if (tmp0.x > 1.0) tmp0.x = 1.0;
  if (tmp0.y < 0.0) tmp0.y = 0.0;
  if (tmp0.y > 1.0) tmp0.y = 1.0;
  if (tmp0.z < 0.0) tmp0.z = 0.0;
  if (tmp0.z > 1.0) tmp0.z = 1.0;
  if (tmp0.w < 0.0) tmp0.w = 0.0;
  if (tmp0.w > 1.0) tmp0.w = 1.0;
  ub.x = round(255.0 * tmp0.x);  /* ub is a ubyte vector */
  ub.y = round(255.0 * tmp0.y);
  ub.z = round(255.0 * tmp0.z);
  ub.w = round(255.0 * tmp0.w);
  /* result obtained by combining raw bits of ub. */
  result.x = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
  result.y = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
  result.z = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
  result.w = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
```

A fragment program will fail to load if it contains a PK4UB instruction that writes its results to a variable declared as "SHORT".

Section 3.11.5.36, RFL: Reflection Vector

The RFL instruction computes the reflection of the second vector operand (the "direction" vector) about the vector specified by the first vector operand (the "axis" vector). Both operands are treated as 3D vectors (the w components are ignored). The result vector is another 3D vector (the "reflected direction" vector). The length of the result vector, ignoring rounding errors, should equal that of the second operand.

```
  axis = VectorLoad(op0);
  direction = VectorLoad(op1);
  tmp.w = (axis.x * axis.x + axis.y * axis.y +
           axis.z * axis.z);
  tmp.x = (axis.x * direction.x + axis.y * direction.y +
           axis.z * direction.z);
  tmp.x = 2.0 * tmp.x;
  tmp.x = tmp.x / tmp.w;
  result.x = tmp.x * axis.x - direction.x;
  result.y = tmp.x * axis.y - direction.y;
  result.z = tmp.x * axis.z - direction.z;
```

A fragment program will fail to load if the w component of the result is enabled in the component write mask.

Section 3.11.5.37, SEQ: Set on Equal

The SEQ instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operand is equal to that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x == tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y == tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z == tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w == tmp1.w) ? 1.0 : 0.0;
```

Section 3.11.5.38, SFL: Set on False

The SFL instruction is a degenerate case of the other "Set on" instructions that sets all components of the result vector to 0.0.

```
  result.x = 0.0;
  result.y = 0.0;
  result.z = 0.0;
  result.w = 0.0;
```

Section 3.11.5.39, SGT: Set on Greater Than

The SGT instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operands is greater than that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x > tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y > tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z > tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w > tmp1.w) ? 1.0 : 0.0;
```

Section 3.11.5.40, SLE: Set on Less Than or Equal

The SLE instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operand is less than or equal to that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x <= tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y <= tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z <= tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w <= tmp1.w) ? 1.0 : 0.0;
```

Section 3.11.5.41, SNE: Set on Not Equal

The SNE instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operand is not equal to that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x != tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y != tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z != tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w != tmp1.w) ? 1.0 : 0.0;
```

Section 3.11.5.42, STR: Set on True

The STR instruction is a degenerate case of the other "Set on" instructions that sets all components of the result vector to 1.0.

```
  result.x = 1.0;
  result.y = 1.0;
  result.z = 1.0;
  result.w = 1.0;
```

Section 3.11.5.43, UP2H: Unpack Two 16-Bit Floats

The UP2H instruction unpacks two 16-bit floats stored together in a 32-bit scalar operand. The first 16-bit float (stored in the 16 least significant bits) is written into the "x" and "z" components of the result vector; the second is written into the "y" and "w" components of the result vector.

This operation undoes the type conversion and packing performed by the PK2H instruction.

```
  tmp = ScalarLoad(op0);
  result.x = (fp16) (RawBits(tmp) & 0xFFFF);
  result.y = (fp16) ((RawBits(tmp) >> 16) & 0xFFFF);
  result.z = (fp16) (RawBits(tmp) & 0xFFFF);
  result.w = (fp16) ((RawBits(tmp) >> 16) & 0xFFFF);
```

A fragment program will fail to load if it contains a UP2H instruction whose operand is a variable declared as "SHORT".

Section 3.11.5.44, UP2US: Unpack Two Unsigned 16-Bit Scalars

The UP2US instruction unpacks two 16-bit unsigned values packed together in a 32-bit scalar operand. The unsigned quantities are encoded where a bit pattern of all '0' bits corresponds to 0.0 and a pattern of all '1' bits corresponds to 1.0. The "x" and "z" components of the result vector are obtained from the 16 least significant bits of the operand; the "y" and "w" components are obtained from the 16 most significant bits.

This operation undoes the type conversion and packing performed by the PK2US instruction.

```
  tmp = ScalarLoad(op0);
  result.x = ((RawBits(tmp) >> 0)  & 0xFFFF) / 65535.0;
  result.y = ((RawBits(tmp) >> 16) & 0xFFFF) / 65535.0;
  result.z = ((RawBits(tmp) >> 0)  & 0xFFFF) / 65535.0;
  result.w = ((RawBits(tmp) >> 16) & 0xFFFF) / 65535.0;
```

A fragment program will fail to load if it contains a UP2S instruction whose operand is a variable declared as "SHORT".

Section 3.11.5.45, UP4B: Unpack Four Signed 8-Bit Values

The UP4B instruction unpacks four 8-bit signed values packed together in a 32-bit scalar operand. The signed quantities are encoded where a bit pattern of all '0' bits corresponds to -128/127 and a pattern of all '1' bits corresponds to +127/127. The "x" component of the result vector is the converted value corresponding to the 8 least significant bits of the operand; the "w" component corresponds to the 8 most significant bits.

This operation undoes the type conversion and packing performed by the PK4B instruction.

```
  tmp = ScalarLoad(op0);
  result.x = (((RawBits(tmp) >> 0) & 0xFF) - 128) / 127.0;
  result.y = (((RawBits(tmp) >> 8) & 0xFF) - 128) / 127.0;
  result.z = (((RawBits(tmp) >> 16) & 0xFF) - 128) / 127.0;
  result.w = (((RawBits(tmp) >> 24) & 0xFF) - 128) / 127.0;
```

A fragment program will fail to load if it contains a UP4B instruction whose operand is a variable declared as "SHORT".

Section 3.11.5.46, UP4UB: Unpack Four Unsigned 8-Bit Scalars

The UP4UB instruction unpacks four 8-bit unsigned values packed together in a 32-bit scalar operand. The unsigned quantities are encoded where a bit pattern of all '0' bits corresponds to 0.0 and a pattern of all '1' bits corresponds to 1.0. The "x" component of the result vector is obtained from the 8 least significant bits of the operand; the "w" component is obtained from the 8 most significant bits.

This operation undoes the type conversion and packing performed by the PK4UB instruction.

```
  tmp = ScalarLoad(op0);
  result.x = ((RawBits(tmp) >> 0)  & 0xFF) / 255.0;
  result.y = ((RawBits(tmp) >> 8)  & 0xFF) / 255.0;
  result.z = ((RawBits(tmp) >> 16) & 0xFF) / 255.0;
  result.w = ((RawBits(tmp) >> 24) & 0xFF) / 255.0;
```

A fragment program will fail to load if it contains a UP4UB instruction whose operand is a variable declared as "SHORT".

Section 3.11.5.47, X2D: 2D Coordinate Transformation

The X2D instruction multiplies the 2D offset vector specified by the "x" and "y" components of the second vector operand by the 2x2 matrix specified by the four components of the third vector operand, and adds the transformed offset vector to the 2D vector specified by the "x" and "y" components of the first vector operand. The first component of the sum is written to the "x" and "z" components of the result; the second component is written to the "y" and "w" components of the result.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  tmp2 = VectorLoad(op2);
  result.x = tmp0.x + tmp1.x * tmp2.x + tmp1.y * tmp2.y;
  result.y = tmp0.y + tmp1.x * tmp2.z + tmp1.y * tmp2.w;
  result.z = tmp0.x + tmp1.x * tmp2.x + tmp1.y * tmp2.y;
  result.w = tmp0.y + tmp1.x * tmp2.z + tmp1.y * tmp2.w;
```

Modify Section, 3.11.6.4 KIL: Kill fragment

Rather than mapping a coordinate set to a color, this function prevents a fragment from receiving any future processing. If any component of its source vector is negative, the processing of this fragment will be discontinued and no further outputs to this fragment will occur. Subsequent stages of the GL pipeline will be skipped for this fragment.

A KIL instruction may be specified using either a vector operand or a condition code test. If a vector operand is specified, the following is performed:

```
  tmp = VectorLoad(op0);
  if ((tmp.x < 0) || (tmp.y < 0) ||
      (tmp.z < 0) || (tmp.w < 0))
  {
      exit;
  }
```

If a condition code is specified, the following is performed:

```
  if (TestCC(rc.c***) || TestCC(rc.*c**) ||
      TestCC(rc.**c*) || TestCC(rc.***c))
  {
     exit;
  }
```

Add Section 3.11.6.5, TXD: Texture Lookup with Derivatives

The TXD instruction takes the first three components of its first vector operand and maps them to s, t, and r. These coordinates are used to sample from the specified texture target on the specified texture image unit in a manner consistent with its parameters.

The level of detail is computed as specified in section 3.8. In this calculation, ds/dx, dt/dx, and dr/dx are given by the x, y, and z components, respectively, of the second vector operand. ds/dy, dt/dy, and dr/dy are given by the x, y, and z components of the third vector operand.

The resulting sample is mapped to RGBA as described in table 3.21 and written to the result vector.

```
  tmp = VectorLoad(op0);
  result = TextureSample(tmp.x, tmp.y, tmp.z, 0.0, op1, op2);
```

Additions to Chapter 4 of the OpenGL 1.2.1 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 1.2.1 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 1.2.1 Specification (State and State Requests)

None.

Additions to Appendix A of the OpenGL 1.2.1 Specification (Invariance)

None.

Additions to the AGL/GLX/WGL Specifications

None.

Dependencies on ARB_fragment_program

This specification is based on a modified version of the grammar published in the ARB_fragment_program specification. This modified grammar (see below) includes a few structural changes to better accommodate new functionality from this and other extensions, but should be functionally equivalent to the ARB_fragment_program grammar.

```
    <program>                ::= <optionSequence> <statementSequence> "END"

    <optionSequence>         ::= <optionSequence> <option>
                               | /* empty */

    <option>                 ::= "OPTION" <optionName> ";"

    <optionName>             ::= "ARB_fog_exp"
                               | "ARB_fog_exp2"
                               | "ARB_fog_linear"
                               | "ARB_precision_hint_fastest"
                               | "ARB_precision_hint_nicest"

    <statementSequence>      ::= <statement> <statementSequence>
                               | /* empty */

    <statement>              ::= <instruction> ";"
                               | <namingStatement> ";"

    <instruction>            ::= <ALUInstruction>
                               | <TexInstruction>

    <ALUInstruction>         ::= <VECTORop_instruction>
                               | <SCALARop_instruction>
                               | <BINSCop_instruction>
                               | <BINop_instruction>
                               | <TRIop_instruction>
                               | <SWZop_instruction>

    <TexInstruction>         ::= <TEXop_instruction>
                               | <KILop_instruction>

    <VECTORop_instruction>   ::= <VECTORop> <instResult> "," <instOperandV>
    <VECTORop>                ::= "ABS"
                                | "FLR"
                                | "FRC"
                                | "LIT"
                                | "MOV"

    <SCALARop_instruction>    ::= <SCALARop> <instResult> "," <instOperandS>

    <SCALARop>                ::= "COS"
                                | "EX2"
                                | "LG2"
                                | "RCP"
                                | "RSQ"
                                | "SCS"
                                | "SIN"

    <BINSCop_instruction>     ::= <BINSCop> <instResult> "," <instOperandS> ","
                                  <instOperandS>

    <BINSCop>                 ::= "POW"

    <BINop_instruction>       ::= <BINop> <instResult> "," <instOperandV> ","
                                  <instOperandV>

    <BINop>                   ::= "ADD"
                                | "DP3"
                                | "DP4"
                                | "DPH"
                                | "DST"
                                | "MAX"
                                | "MIN"
                                | "MUL"
                                | "SGE"
                                | "SLT"
                                | "SUB"
                                | "XPD"

    <TRIop_instruction>       ::= <TRIop> <instResult> "," <instOperandV> ","
                                  <instOperandV> "," <instOperandV>

    <TRIop>                   ::= "CMP"
                                | "MAD"
                                | "LRP"

    <SWZop_instruction>       ::= <SWZop> <instResult> "," <instOperandVNS> ","
                                  <extendedSwizzle>

    <SWZop>                   ::= "SWZ"

    <TEXop_instruction>       ::= <TEXop> <instResult> "," <instOperandV> ","
                                  <texTarget>

    <TEXop>                   ::= "TEX"
                                | "TXP"
                                | "TXB"

    <KILop_instruction>       ::= <KILop> <killCond>
    <KILop>                 ::= "KIL"

    <texTarget>             ::= <texImageUnit> "," <texTargetType>

    <texImageUnit>          ::= "texture" <optTexImageUnitNum>

    <optTexImageUnitNum>    ::= /* empty */
                              | "[" <texImageUnitNum> "]"

    <texImageUnitNum>       ::= <integer>
                                /*[0,MAX_TEXTURE_IMAGE_UNITS_ARB-1]*/

    <texTargetType>         ::= "1D"
                              | "2D"
                              | "3D"
                              | "CUBE"
                              | "RECT"

    <killCond>              ::= <instOperandV>

    <instOperandV>          ::= <instOperandBaseV>

    <instOperandBaseV>      ::= <optSign> <attribUseV>
                              | <optSign> <tempUseV>
                              | <optSign> <paramUseV>

    <instOperandS>          ::= <instOperandBaseS>

    <instOperandBaseS>      ::= <optSign> <attribUseS>
                              | <optSign> <tempUseS>
                              | <optSign> <paramUseS>

    <instOperandVNS>        ::= <attribUseVNS>
                              | <tempUseVNS>
                              | <paramUseVNS>

    <instResult>            ::= <instResultBase>

    <instResultBase>        ::= <tempUseW>
                              | <resultUseW>

    <namingStatement>       ::= <ATTRIB_statement>
                              | <PARAM_statement>
                              | <TEMP_statement>
                              | <OUTPUT_statement>
                              | <ALIAS_statement>

    <ATTRIB_statement>      ::= "ATTRIB" <establishName> "=" <attribUseD>

    <PARAM_statement>       ::= <PARAM_singleStmt>
                              | <PARAM_multipleStmt>

    <PARAM_singleStmt>      ::= "PARAM" <establishName> <paramSingleInit>

    <PARAM_multipleStmt>    ::= "PARAM" <establishName> "[" <optArraySize> "]"
                                <paramMultipleInit>
    <optArraySize>          ::= /* empty */
                              | <integer> /* [1,MAX_PROGRAM_PARAMETERS_ARB]*/

    <paramSingleInit>       ::= "=" <paramUseDB>

    <paramMultipleInit>     ::= "=" "{" <paramMultInitList> "}"

    <paramMultInitList>     ::= <paramUseDM>
                              | <paramUseDM> "," <paramMultInitList>

    <TEMP_statement>        ::= "TEMP" <varNameList>

    <OUTPUT_statement>      ::= "OUTPUT" <establishName> "=" <resultUseD>

    <ALIAS_statement>       ::= "ALIAS" <establishName> "=" <establishedName>

    <establishedName>       ::= <tempVarName>
                              | <addrVarName>
                              | <attribVarName>
                              | <paramArrayVarName>
                              | <paramSingleVarName>
                              | <resultVarName>

    <varNameList>           ::= <establishName>
                              | <establishName> "," <varNameList>

    <establishName>         ::= <identifier>

    <attribUseV>            ::= <attribBasic> <swizzleSuffix>
                              | <attribVarName> <swizzleSuffix>
                              | <attribColor> <swizzleSuffix>
                              | <attribColor> "." <colorType> <swizzleSuffix>

    <attribUseS>            ::= <attribBasic> <scalarSuffix>
                              | <attribVarName> <scalarSuffix>
                              | <attribColor> <scalarSuffix>
                              | <attribColor> "." <colorType> <scalarSuffix>

    <attribUseVNS>          ::= <attribBasic>
                              | <attribVarName>
                              | <attribColor>
                              | <attribColor> "." <colorType>

    <attribUseD>            ::= <attribBasic>
                              | <attribColor>
                              | <attribColor> "." <colorType>

    <attribBasic>           ::= "fragment" "." <attribFragBasic>

    <attribFragBasic>       ::= "texcoord" <optTexCoordNum>
                              | "fogcoord"
                              | "position"

    <attribColor>           ::= "fragment" "." "color"
    <paramUseV>              ::= <paramSingleVarName> <swizzleSuffix>
                              | <paramArrayVarName> "[" <arrayMem> "]"
                                <swizzleSuffix>
                              | <stateSingleItem> <swizzleSuffix>
                              | <programSingleItem> <swizzleSuffix>
                              | <constantVector> <swizzleSuffix>
                              | <constantScalar> <swizzleSuffix>

    <paramUseS>              ::= <paramSingleVarName> <scalarSuffix>
                              | <paramArrayVarName> "[" <arrayMem> "]"
                                <scalarSuffix>
                              | <stateSingleItem> <scalarSuffix>
                              | <programSingleItem> <scalarSuffix>
                              | <constantVector> <scalarSuffix>
                              | <constantScalar> <scalarSuffix>

    <paramUseVNS>            ::= <paramSingleVarName>
                              | <paramArrayVarName> "[" <arrayMem> "]"
                              | <stateSingleItem>
                              | <programSingleItem>
                              | <constantVector>
                              | <constantScalar>

    <paramUseDB>             ::= <stateSingleItem>
                              | <programSingleItem>
                              | <constantVector>
                              | <signedConstantScalar>

    <paramUseDM>             ::= <stateMultipleItem>
                              | <programMultipleItem>
                              | <constantVector>
                              | <signedConstantScalar>

    <stateMultipleItem>      ::= <stateSingleItem>
                              | "state" "." <stateMatrixRows>

    <stateSingleItem>        ::= "state" "." <stateMaterialItem>
                              | "state" "." <stateLightItem>
                              | "state" "." <stateLightModelItem>
                              | "state" "." <stateLightProdItem>
                              | "state" "." <stateFogItem>
                              | "state" "." <stateMatrixRow>
                              | "state" "." <stateTexEnvItem>
                              | "state" "." <stateDepthItem>

    <stateMaterialItem>      ::= "material" "." <stateMatProperty>
                              | "material" "." <faceType> "."
                                <stateMatProperty>

    <stateMatProperty>       ::= "ambient"
                              | "diffuse"
                              | "specular"
                              | "emission"
                              | "shininess"

    <stateLightItem>         ::= "light" "[" <stateLightNumber> "]" "."
                                <stateLightProperty>
    <stateLightProperty>    ::= "ambient"
                              | "diffuse"
                              | "specular"
                              | "position"
                              | "attenuation"
                              | "spot" "." <stateSpotProperty>
                              | "half"

    <stateSpotProperty>     ::= "direction"

    <stateLightModelItem>   ::= "lightmodel" <stateLModProperty>

    <stateLModProperty>     ::= "." "ambient"
                              | "." "scenecolor"
                              | "." <faceType> "." "scenecolor"

    <stateLightProdItem>    ::= "lightprod" "[" <stateLightNumber> "]" "."
                                <stateLProdProperty>
                              | "lightprod" "[" <stateLightNumber> "]" "."
                                <faceType> "." <stateLProdProperty>

    <stateLProdProperty>    ::= "ambient"
                              | "diffuse"
                              | "specular"

    <stateLightNumber>      ::= <integer> /* [0,MAX_LIGHTS-1] */

    <stateFogItem>          ::= "fog" "." <stateFogProperty>

    <stateFogProperty>      ::= "color"
                              | "params"

    <stateMatrixRows>       ::= <stateMatrixItem>
                              | <stateMatrixItem> "." <stateMatModifier>
                              | <stateMatrixItem> "." "row" "["
                                <stateMatrixRowNum> ".." <stateMatrixRowNum>
                                "]"
                              | <stateMatrixItem> "." <stateMatModifier> "."
                                "row" "[" <stateMatrixRowNum> ".."
                                <stateMatrixRowNum> "]"

    <stateMatrixRow>        ::= <stateMatrixItem> "." "row" "["
                                <stateMatrixRowNum> "]"
                              | <stateMatrixItem> "." <stateMatModifier> "."
                                "row" "[" <stateMatrixRowNum> "]"

    <stateMatrixItem>       ::= "matrix" "." <stateMatrixName>

    <stateMatModifier>      ::= "inverse"
                              | "transpose"
                              | "invtrans"
    <stateMatrixName>       ::= "modelview" <stateOptModMatNum>
                              | "projection"
                              | "mvp"
                              | "texture" <optTexCoordNum>
                              | "palette" "[" <statePaletteMatNum> "]"
                              | "program" "[" <stateProgramMatNum> "]"

    <stateMatrixRowNum>     ::= <integer> /* [0,3] */

    <stateOptModMatNum>     ::= /* empty */
                              | "[" <stateModMatNum> "]"

    <stateModMatNum>        ::= <integer> /*[0,MAX_VERTEX_UNITS_ARB-1]*/

    <statePaletteMatNum>    ::= <integer> /*[0,MAX_PALETTE_MATRICES_ARB-1]*/

    <stateProgramMatNum>    ::= <integer> /*[0,MAX_PROGRAM_MATRICES_ARB-1]*/

    <stateTexEnvItem>       ::= "texenv" <optLegacyTexUnitNum> "."
                                <stateTexEnvProperty>

    <stateTexEnvProperty>   ::= "color"

    <stateDepthItem>        ::= "depth" "." <stateDepthProperty>

    <stateDepthProperty>    ::= "range"

    <programSingleItem>     ::= <progEnvParam>
                              | <progLocalParam>

    <programMultipleItem>   ::= <progEnvParams>
                              | <progLocalParams>

    <progEnvParams>         ::= "program" "." "env" "[" <progEnvParamNums> "]"

    <progEnvParamNums>      ::= <progEnvParamNum>
                              | <progEnvParamNum> ".." <progEnvParamNum>

    <progEnvParam>          ::= "program" "." "env" "[" <progEnvParamNum> "]"

    <progLocalParams>       ::= "program" "." "local" "[" <progLocalParamNums>
                                "]"

    <progLocalParamNums>    ::= <progLocalParamNum>
                              | <progLocalParamNum> ".." <progLocalParamNum>

    <progLocalParam>        ::= "program" "." "local" "[" <progLocalParamNum>
                                "]"

    <progEnvParamNum>       ::= <integer>
                                /*[0,MAX_PROGRAM_ENV_PARAMETERS_ARB-1]*/

    <progLocalParamNum>     ::= <integer>
                                /*[0,MAX_PROGRAM_LOCAL_PARAMETERS_ARB-1]*/

    <constantVector>        ::= "{" <constantVectorList> "}"
    <constantVectorList>    ::= <signedConstantScalar>
                              | <signedConstantScalar> ","
                                <signedConstantScalar>
                              | <signedConstantScalar> ","
                                <signedConstantScalar> ","
                                <signedConstantScalar>
                              | <signedConstantScalar> ","
                                <signedConstantScalar> ","
                                <signedConstantScalar> ","
                                <signedConstantScalar>

    <signedConstantScalar>  ::= <optSign> <constantScalar>

    <constantScalar>        ::= <floatConstant>

    <floatConstant>         ::= <float>

    <tempUseV>              ::= <tempVarName> <swizzleSuffix>

    <tempUseS>              ::= <tempVarName> <scalarSuffix>

    <tempUseVNS>            ::= <tempVarName>

    <tempUseW>              ::= <tempVarName> <optWriteMask>

    <resultUseW>            ::= <resultBasic> <optWriteMask>
                              | <resultVarName> <optWriteMask>

    <resultUseD>            ::= <resultBasic>

    <resultBasic>           ::= "result" "." <resultFragBasic>

    <resultFragBasic>       ::= "color" <resultOptColorNum>
                              | "depth"

    <resultOptColorNum>     ::= /* empty */

    <arrayMem>              ::= <arrayMemAbs>

    <arrayMemAbs>           ::= <integer>

    <optWriteMask>          ::= /* empty */
                              | <xyzwMask>
                              | <rgbaMask>
    <xyzwMask>              ::= "." "x"
                              | "." "y"
                              | "." "xy"
                              | "." "z"
                              | "." "xz"
                              | "." "yz"
                              | "." "xyz"
                              | "." "w"
                              | "." "xw"
                              | "." "yw"
                              | "." "xyw"
                              | "." "zw"
                              | "." "xzw"
                              | "." "yzw"
                              | "." "xyzw"

    <rgbaMask>              ::= "." "r"
                              | "." "g"
                              | "." "rg"
                              | "." "b"
                              | "." "rb"
                              | "." "gb"
                              | "." "rgb"
                              | "." "a"
                              | "." "ra"
                              | "." "ga"
                              | "." "rga"
                              | "." "ba"
                              | "." "rba"
                              | "." "gba"
                              | "." "rgba"

    <swizzleSuffix>         ::= /* empty */
                              | "." <component>
                              | "." <xyzwComponent> <xyzwComponent>
                                <xyzwComponent> <xyzwComponent>
                              | "." <rgbaComponent> <rgbaComponent>
                                <rgbaComponent> <rgbaComponent>

    <extendedSwizzle>       ::= <extSwizComp> "," <extSwizComp> ","
                                <extSwizComp> "," <extSwizComp>

    <extSwizComp>           ::= <optSign> <xyzwExtSwizSel>
                              | <optSign> <rgbaExtSwizSel>

    <xyzwExtSwizSel>        ::= "0"
                              | "1"
                              | <xyzwComponent>

    <rgbaExtSwizSel>        ::= <rgbaComponent>

    <scalarSuffix>          ::= "." <component>

    <component>             ::= <xyzwComponent>
                              | <rgbaComponent>
    <xyzwComponent>         ::= "x"
                              | "y"
                              | "z"
                              | "w"

    <rgbaComponent>         ::= "r"
                              | "g"
                              | "b"
                              | "a"

    <optSign>               ::= /* empty */
                              | "-"
                              | "+"

    <faceType>              ::= "front"
                              | "back"

    <colorType>             ::= "primary"
                              | "secondary"

    <optTexCoordNum>        ::= /* empty */
                              | "[" <texCoordNum> "]"

    <texCoordNum>           ::= <integer> /*[0,MAX_TEXTURE_COORDS_ARB-1]*/

    <optLegacyTexUnitNum>   ::= /* empty */
                              | "[" <legacyTexUnitNum> "]"

    <legacyTexUnitNum>      ::= <integer> /*[0,MAX_TEXTURE_UNITS-1]*/
```

The <integer>, <float>, and <identifier> grammar rules match integer constants, floating point constants, and identifier names as described in the ARB_vertex_program specification. The <float> grammar rule here is identical to the <floatConstant> grammar rule in ARB_vertex_program.

The grammar rules <tempVarName>, <addrVarName>, <attribVarName>, <paramArrayVarName>, <paramSingleVarName>, <resultVarName> refer to the names of temporary, address register, attribute, program parameter array, program parameter, and result variables declared in the program text.

GLX Protocol

None.

Errors

None.

New State

None.

Revision History

```
    Rev.  Date      Author   Changes
    ----  --------  -------  ------------------------------------------
    4     05/27/05  pbrown   Removed required NV_fragment_program dependency;
                             that extension actually isn't needed although the
                             functionality it provides obviously is.

    3     07/08/04  pbrown   Fixed entries for KIL and RFL in the opcode
                             table.

    2     05/16/04  pbrown   Documented terminals in modified fragment
                             program grammar.

    1     --------  pbrown   Internal pre-release revisions.
```

Name

    NV_fragment_program2

Name Strings

    GL_NV_fragment_program2

Status

    Shipping.

Version

```
    Last Modified:      08/04/2004
    NVIDIA Revision:    8
```

Number

    304

Dependencies

    ARB_fragment_program is required.
    NV_fragment_program_option is required.

Overview

```
    This extension, like the NV_fragment_program_option extension, provides
    additional fragment program functionality to extend the standard
    ARB_fragment_program language and execution environment.  ARB programs
    wishing to use this added functionality need only add:

        OPTION NV_fragment_program2;

    to the beginning of their fragment programs.

    New functionality provided by this extension, above and beyond that
    already provided by the NV_fragment_program_option extension, includes:

      * structured branching support, including data-dependent IF tests, loops
        supporting a fixed number of iterations, and a data-dependent loop
        exit instruction (BRK),

      * subroutine calls,

      * instructions to perform vector normalization, divide vector components
        by a scalar, and perform two-component dot products (with or without a
        scalar add),

      * an instruction to perform a texture lookup with an explicit LOD,

      * a loop index register for indirect access into the texture coordinate
        attribute array, and

      * a facing attribute that indicates whether the fragment is generated
        from a front- or back-facing primitive.
```

Issues

** Should this extension expose projective forms of the LOD-modifying texture instructions?*

RESOLVED: No. The user can manually add a DIV instruction to achieve the same effect.

** Should this extension expose precision explicitly?*

RESOLVED: Only for storage using the SHORT TEMP and LONG TEMP syntax (similar to NV_fragment_program_option).

** How are resources (such as registers and condition codes) scoped?*

RESOLVED: All resources are globally scoped. This means that if, for instance, a subroutine modifies a condition code, that modification effects both the caller and the callee.

** How is the scope determined for instructions required to be within a specific loop construct?*

RESOLVED: The scope is determined statically at compile time. This means that calling BRK and using A0 from a subroutine called within a loop is a compile error.

New Procedures and Functions

None.

New Tokens

Accepted by the <pname> parameter of GetProgramivARB:

```
MAX_PROGRAM_EXEC_INSTRUCTIONS_NV               0x88F4
MAX_PROGRAM_CALL_DEPTH_NV                      0x88F5
MAX_PROGRAM_IF_DEPTH_NV                        0x88F6
MAX_PROGRAM_LOOP_DEPTH_NV                      0x88F7
MAX_PROGRAM_LOOP_COUNT_NV                      0x88F8
```

Additions to Chapter 2 of the OpenGL 1.2.1 Specification (OpenGL Operation)

None.

Additions to Chapter 3 of the OpenGL 1.2.1 Specification (Rasterization)

Modify Section 3.11 of ARB_fragment_program (Fragment Program):

Delete the sentence referring to the lack of branching or looping.

Modify Section 3.11.2 of ARB_fragment_program (Fragment Program Grammar and Restrictions):

(mostly add to existing grammar rules, as extended by NV_fragment_program_option)

```
    <optionName>              ::= "NV_fragment_program2"

    <statement>               ::= <branchLabel> ":"

    <instruction>             ::= <FlowInstruction>

    <ALUInstruction>          ::= <VECSCAop_instruction>

    <FlowInstruction>         ::= <BRAop_instruction>
                                | <FLOWCCop_instruction>
                                | <IFop_instruction>
                                | <LOOPop_instruction>
                                | <ENDFLOWop_instruction>

    <VECTORop>                ::= "NRM"

    <VECSCAop_instruction>    ::= <VECSCAop> <instResult> "," <instOperandV> ","
                                  <instOperandS>

    <VECSCAop>                ::= "DIV"

    <BINop>                   ::= "DP2"

    <TRIop>                   ::= "DP2A"

    <TEXop>                   ::= "TXL"

    <BRAop_instruction>       ::= <BRAop> <branchLabel> <optBranchCond>

    <BRAop>                   ::= "CAL"

    <FLOWCCop_instruction>    ::= <FLOWCCop> <optBranchCond>

    <FLOWCCop>                ::= "RET"
                                | "BRK"

    <IFop_instruction>        ::= <IFop> <ccTest>

    <IFop>                    ::= "IF"

    <LOOPop_instruction>      ::= <LOOPop> <instOperandV>

    <LOOPop>                  ::= "LOOP"
                                | "REP"

    <ENDFLOWop_instruction>   ::= <ENDFLOWop>

    <ENDFLOWop>               ::= "ELSE"
                                | "ENDIF"
                                | "ENDLOOP"
                                | "ENDREP"

    <optBranchCond>           ::= /* empty */
                                | <ccMask>

    <branchLabel>             ::= <identifier>
```

```
<attribFragBasic>       ::= "texcoord" "[" <arrayMemRel> "]"
                          | "facing"

<arrayMemRel>           ::= <addrUseS> <arrayMemRelOffset>

<arrayMemRelOffset>     ::= /* empty */
                          | "+" <addrRegPosOffset>

<addrRegPosOffset>      ::= <integer> from 0 to 9

<addrUseS>              ::= <addrVarName> <scalarAddrSuffix>

<scalarAddrSuffix>      ::= "." <addrComponent>

<addrComponent>         ::= "x"
```

Note: This extension provides a pre-defined address register (A0) that matches the <addrVarName> grammar rule and can be used as a loop counter (Section 3.11.3.Y). It is not possible to declare additional address register variables.

Modify Section 3.11.3.1, Fragment Attributes

(add new bindings to binding table)

```
  Fragment Attribute Binding  Components  Underlying State
  --------------------------  ----------  ----------------------------
  ...
  fragment.texcoord[A0.x+n]   (s,t,r,q)   indexed texture coordinate
  fragment.facing             (f,0,0,1)   fragment facing
```

If a fragment attribute binding matches "fragment.texcoord[A0.x+n]", a texture coordinate number <c> is computed by adding the current value of the "A0.x" address register (the loop index -- Section 3.11.3.Y) and <n>. The "x", "y", "z", and "w" components of the fragment attribute variable are filled with the "s", "t", "r", and "q" components, respectively, of the fragment texture coordinates for texture coordinate set <c>. If <c> is negative or greater than or equal to MAX_TEXTURE_COORDS_ARB, the fragment attribute variable is undefined.

If a fragment attribute binding matches "fragment.facing", the "x" component of the fragment attribute variable is filled with +1.0 or -1.0, depending on the orientation of the primitive producing the fragment. If the fragment is generated by a back-facing polygon (including point- and line-mode polygons), the facing is -1.0; otherwise, the facing is +1.0. The "y", "z", and "w" coordinates are filled with 0, 0, and 1, respectively.

Add New Section 3.11.3.Y, Fragment Program Address Register (insert after Section 3.11.3.X, Condition Code Register)

Fragment program address register variables are a set of four-component signed integer vectors where only the "x" component of the address registers is currently accessible. Address registers are used as indices when performing relative addressing in the "fragment.texcoord" attribute array (section 3.11.3.1).

Fragment program address registers can not be declared in a fragment program. There is only a single built-in address register, "A0.x" (loop index), which is available inside LOOP/ENDLOOP blocks. A fragment program that accesses A0.x outside a LOOP/ENDLOOP block will fail to load.

A0.x is initialized in by the LOOP instruction and updated by the ENDLOOP instruction. When LOOP blocks are nested, each block has its own value for A0.x, but only the A0.x value for the innermost block can be used. The value of A0.x is clamped to be greater than or equal to 0.

Modify Section 3.11.4, Fragment Program Execution Environment

(modify instruction table) There are sixty-seven fragment program instructions....

Instr.	Modifiers R	H	X	C	S	Inputs	Output	Description
ABS	X	X	X	X	X	v	v	absolute value
ADD	X	X	X	X	X	v,v	v	add
BRK	-	-	-	-	-	c	-	break out of loop instruction
CAL	-	-	-	-	-	c	-	subroutine call
CMP	-	-	-	X	X	v,v,v	v	compare
COS	X	X	-	X	X	s	ssss	cosine with reduction to [-PI,PI]
DDX	X	X	-	X	X	v	v	partial derivative relative to X
DDY	X	X	-	X	X	v	v	partial derivative relative to Y
DIV	X	X	-	X	X	v,s	v	divide vector components by scalar
DP2	X	X	X	X	X	v,v	ssss	2-component dot product
DP2A	X	X	X	X	X	v,v,v	ssss	2-comp. dot product w/scalar add
DP3	X	X	X	X	X	v,v	ssss	3-component dot product
DP4	X	X	X	X	X	v,v	ssss	4-component dot product
DPH	X	X	X	X	X	v,v	ssss	homogeneous dot product
DST	X	X	-	X	X	v,v	v	distance vector
ELSE	-	-	-	-	-	-	-	start if test else block
ENDIF	-	-	-	-	-	-	-	end if test block
ENDLOOP	-	-	-	-	-	-	-	end of loop block
ENDREP	-	-	-	-	-	-	-	end of repeat block
EX2	X	X	-	X	X	s	ssss	exponential base 2
FLR	X	X	X	X	X	v	v	floor
FRC	X	X	X	X	X	v	v	fraction
IF	-	-	-	-	-	c	-	start of if test block
KIL	-	-	-	-	-	v or c	-	kill fragment
LG2	X	X	-	X	X	s	ssss	logarithm base 2
LIT	X	X	-	X	X	v	v	compute light coefficients
LOOP	-	-	-	-	-	v	-	start of loop block
LRP	X	X	X	X	X	v,v,v	v	linear interpolation
MAD	X	X	X	X	X	v,v,v	v	multiply and add
MAX	X	X	X	X	X	v,v	v	maximum
MIN	X	X	X	X	X	v,v	v	minimum
MOV	X	X	X	X	X	v	v	move
MUL	X	X	X	X	X	v,v	v	multiply
NRM	X	X	-	X	X	v	v	normalize 3-component vector
PK2H	-	-	-	-	-	v	ssss	pack two 16-bit floats
PK2US	-	-	-	-	-	v	ssss	pack two unsigned 16-bit scalars
PK4B	-	-	-	-	-	v	ssss	pack four signed 8-bit scalars
PK4UB	-	-	-	-	-	v	ssss	pack four unsigned 8-bit scalars

Instr.	Modifiers R	H	X	C	S	Inputs	Output	Description
POW	X	X	-	X	X	s,s	ssss	exponentiate
RCP	X	X	-	X	X	s	ssss	reciprocal
REP	-	-	-	-	-	v	-	start of repeat block
RET	-	-	-	-	-	c	-	subroutine return
RFL	X	X	-	X	X	v,v	v	reflection vector
RSQ	X	X	-	X	X	s	ssss	reciprocal square root
SCS	X	X	-	X	X	s	ss--	sine/cosine without reduction
SEQ	X	X	X	X	X	v,v	v	set on equal
SFL	X	X	X	X	X	v,v	v	set on false
SGE	X	X	X	X	X	v,v	v	set on greater than or equal
SGT	X	X	X	X	X	v,v	v	set on greater than
SIN	X	X	-	X	X	s	ssss	sine with reduction to [-PI,PI]
SLE	X	X	X	X	X	v,v	v	set on less than or equal
SLT	X	X	X	X	X	v,v	v	set on less than
SNE	X	X	X	X	X	v,v	v	set on not equal
STR	X	X	X	X	X	v,v	v	set on true
SUB	X	X	X	X	X	v,v	v	subtract
SWZ	X	X	-	X	X	v	v	extended swizzle
TEX	-	-	-	X	X	v	v	texture sample
TXB	-	-	-	X	X	v	v	texture sample with bias
TXD	-	-	-	X	X	v,v,v	v	texture sample w/partials
TXL	-	-	-	X	X	v	v	texture same w/explicit LOD
TXP	-	-	-	X	X	v	v	texture sample with projection
UP2H	-	-	-	X	X	s	v	unpack two 16-bit floats
UP2US	-	-	-	X	X	s	v	unpack two unsigned 16-bit scalars
UP4B	-	-	-	X	X	s	v	unpack four signed 8-bit scalars
UP4UB	-	-	-	X	X	s	v	unpack four unsigned 8-bit scalars
X2D	X	X	-	X	X	v,v,v	v	2D coordinate transformation
XPD	X	X	-	X	X	v,v	v	cross product

Table X.5: Summary of fragment program instructions. The columns "R", "H", "X", "C", and "S" indicate whether the "R", "H", or "X" precision modifiers, the C condition code update modifier, and the "_SAT"/"_SSAT" saturation modifiers, respectively, are supported for the opcode. In the input/output columns, "v" indicates a floating-point vector input or output, "s" indicates a floating-point scalar input, "ssss" indicates a scalar output replicated across a 4-component result vector, "ss--" indicates two scalar outputs in the first two components, and "c" indicates a condition code test. Instructions describe as "texture sample" also specify a texture image unit identifier and a texture target.

Modify Section 3.11.4.3, Fragment Program Destination Register Update

(modify saturation discussion) If the instruction opcode has the "_SAT" suffix, requesting saturated result vectors, each component of the result vector is clamped to the range [0,1] before updating the destination register. If the instruction opcode has the "_SSAT" suffix, requesting signed saturation, each component of the result vector is clamped to the range [-1,1] before updating the destination register.

Add Section 3.11.4.X, Fragment Program Branching (before Section 3.11.4.4, Fragment Program Result Processing)

Fragment programs support a limited model of branching. Fragment programs can specify one of several types of instruction blocks: IF/ELSE/ENDIF blocks, LOOP/ENDLOOP blocks, and REP/ENDREP blocks. Examples include the following:

```
  LOOP {5, 0, 1};     # 5 iterations with loop index at 0,1,2,3,4
  ADD R0, R0, R1;
  ENDLOOP;

  REP repCount;
  ADD R0, R0, R1;
  ENDREP;

  MOVC CC, R0;
  IF GT.x;
    MOV R0, R1;  # executes if R0.x > 0
  ELSE;
    MOV R0, R2;  # executes if R0.x <= 0
  ENDIF;
```

Instruction blocks may be nested -- for example, a LOOP block may be contained inside an IF/ELSE/ENDIF block. In all cases, each instruction block must be terminated with the appropriate instruction (ENDIF for IF, ENDLOOP for LOOP, ENDREP for REP). Nested instruction blocks must be wholly contained within a block -- if a LOOP instruction is found between an IF and ELSE instruction, the ENDLOOP must also be present between the IF and ELSE. A fragment program will fail to load if any instruction block is terminated by an incorrect instruction or is not terminated before the block containing it.

IF/ELSE/ENDIF blocks evaluate a condition to determine which instructions to execute. If the condition is true, all instructions between the IF and ELSE are executed. If the condition is false, all instructions between the ELSE and ENDIF are executed. The ELSE instruction is optional. If the ELSE is omitted, all instructions between the IF and ENDIF are executed if the condition is true, or skipped if the condition is false. A limited amount of nesting is supported -- a fragment program will fail to load if an IF instruction is nested inside MAX_PROGRAM_IF_DEPTH_NV or more IF/ELSE/ENDIF blocks.

The condition of an IF test is specified by the <ccTest> grammar rule and may depend on the contents of the condition code register. Branch conditions are evaluated by evaluating a condition code write mask in exactly the same manner as done for register writes (section 2.14.2.2). If any of the four components of the condition code write mask are enabled, the branch is taken and execution continues with the instruction following the label specified in the instruction. Otherwise, the instruction is ignored and fragment program execution continues with the next instruction. In the following example code,

```
    MOVC CC, c[0];         # c[0]=(-2, 0, 2, NaN), CC gets (LT,EQ,GT,UN)
    CAL label1 (LT.xyzw);  # call taken
    CAL label2 (LT.wyzw);  # call not taken
```

the first CAL instruction loads a condition code of (LT,EQ,GT,UN) while the second CAL instruction loads a condition code of (UN,EQ,GT,UN). The first call will be made because the "x" component evaluates to LT; the second call will not be made because no component evaluates to LT.

LOOP/ENDLOOP and REP/ENDREP blocks involve a loop counter that indicates the number of times the instructions between the LOOP/REP and ENDLOOP/ENDREP are executed. Looping blocks have a number of significant limitations. First, the loop counter can not be computed at run time; it must be specified as a program parameter. Second, the number of loop iterations is limited to the value MAX_PROGRAM_LOOP_COUNT_NV, which must be at least 255. Third, only a limited amount of nesting is supported -- a fragment program will fail to load if a LOOP or REP instruction is nested inside MAX_PROGRAM_LOOP_DEPTH_NV or more LOOP/ENDLOOP or REP/ENDREP blocks.

The BRK instruction is available to terminate a loop block early. A BRK instruction can be conditional; the condition is evaluated in the same manner as the condition of an IF instruction, and the loop is terminated if the condition is true. A fragment program will fail to load if it contains a BRK instruction that is not nested inside a LOOP/ENDLOOP or REP/ENDREP block.

Fragment programs can contain one or more instruction labels, matching the grammar rule <branchLabel>. An instruction label can be referred to explicitly in subroutine call (CAL) instructions. Instruction labels can be used at any point in the body of a program, and can be used in instructions before being defined in the program string. Instruction labels can be defined anywhere in the program, except inside an IF/ELSE/ENDIF, LOOP/ENDLOOP, or REP/ENDREP instruction block. A fragment program will fail to load if it contains an instruction label inside an instruction block.

Fragment programs can also specify subroutine calls. When a subroutine call (CAL) instruction is executed, a reference to the instruction immediately following the CAL instruction is pushed onto the call stack. When a subroutine return (RET) instruction is executed, an instruction reference is popped off the call stack and program execution continues with the popped instruction. A fragment program will terminate if a CAL instruction is executed with MAX_PROGRAM_CALL_DEPTH_NV entries already in the call stack or if a RET instruction is executed with an empty call stack. Subroutine calls may be conditional; the condition is specified by the <optBranchCond> grammar rule and evaluated in the same way as the condition of the IF instruction. If no condition is specified, it is as though "(TR)" were specified -- the branch is unconditional.

If a fragment program has an instruction label "main", program execution begins with the instruction immediately following the instruction label. Otherwise, program execution begins with the first instruction of the program. Instructions will be executed sequentially in the order specified in the program, although branch instructions will affect the instruction execution order, as described above. A fragment program will terminate after executing a RET instruction with an empty call stack. A fragment program will also terminate after executing the last instruction in the program, unless that instruction was a taken branch.

A fragment program will fail to load if an instruction refers to a label that is not defined in the program string.

A fragment program will terminate abnormally if a subroutine call instruction produces a call stack overflow. Additionally, a fragment program will terminate abnormally after executing MAX_PROGRAM_EXEC_INSTRUCTIONS instructions to prevent hangs caused by infinite loops in the program.

When a fragment program terminates, normally or abnormally, it will emit a fragment whose attributes are taken from the final values of the fragment program result variables (section 3.11.3.4).

Add to Section 3.11.4.5 of ARB_fragment_program (Fragment Program Options):

Section 3.11.4.5.3, NV_fragment_program2 Option

If a fragment program specifies the "NV_fragment_program2" option, the ARB_fragment_program grammar and execution environment are extended to take advantage of all the features of the "NV_fragment_program" option, plus the following features:

* structured branching support, including data-dependent IF tests, loops supporting a fixed number of iterations, and a data-dependent loop exit instruction (BRK),

* subroutine calls,

* several new instructions:

  * NRM -- vector normalization
  * DIV -- divide vector components by a scalar
  * DP2 -- two-component dot product
  * DP2A -- two-component dot product with scalar add
  * TXL -- texture lookup with explicit LOD specified
  * IF/ELSE/ENDIF -- conditional execution blocks
  * REP/ENDREP -- loop block
  * LOOP/ENDLOOP -- loop block using index register
  * BRK -- break out of loop block
  * CAL -- subroutine call
  * RET -- subroutine return

* a loop index register inside LOOP/ENDLOOP blocks that can be used for indirect access into the texture coordinate attribute array, and

* a facing attribute that indicates whether the fragment is generated from a front- or back-facing primitive.

Modify Section 3.11.5, Fragment Program ALU Instruction Set

Section 3.11.5.48, DIV: Divide (Vector Components by Scalar)

The DIV instruction divides each component of the first vector operand by the second scalar operand to produce a 4-component result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = ScalarLoad(op1);
  result.x = tmp0.x / tmp1;
  result.y = tmp0.y / tmp1;
  result.z = tmp0.z / tmp1;
  result.w = tmp0.w / tmp1;
```

This instruction may not produce results identical to a RCP/MUL instruction sequence.

Section 3.11.5.49, DP2: 2-Component Dot Product

The DP2 instruction computes a two-component dot product of the two operands (using the first two components) and replicates the dot product to all four components of the result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  dot = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y);
  result.x = dot;
  result.y = dot;
  result.z = dot;
  result.w = dot;
```

Section 3.11.5.50, DP2A: 2-Component Dot Product w/Scalar Add

The DP2 instruction computes a two-component dot product of the two operands (using the first two components), adds the x component of the third operand, and replicates the result to all four components of the result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  tmp2 = VectorLoad(op2);
  dot = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) + tmp2.x;
  result.x = dot;
  result.y = dot;
  result.z = dot;
  result.w = dot;
```

Section 3.11.5.51, NRM: 3-Component Vector Normalize

The NRM instruction normalizes the vector given by the x, y, and z components of the vector operand to produce the x, y, and z components of the result vector. The w component of the result is undefined.

```
  tmp = VectorLoad(op0);
  scale = ApproxRSQ(tmp.x * tmp.x + tmp.y * tmp.y + tmp.z * tmp.z);
  result.x = tmp.x * scale;
  result.y = tmp.y * scale;
  result.z = tmp.z * scale;
  result.w = undefined;
```

Note that the normalization uses an approximate scale and may be carried at lower precision than a corresponding sequence of DP3, RSQ, and MUL instructions.

Add Section 3.11.6.6, TXL: Texture Lookup with Explicit LOD

The TXL instruction takes the x, y, and z components of the vector operand and maps them to s, t, and r, respectively. These coordinates are used to sample from the specified texture target on the specified texture image unit in a manner consistent with its parameters.

The level of detail is computed as specified in section 3.8.8, except that rho(x,y) is given by 2^w, where w is the w component of the vector operand.

The resulting sample is mapped to RGBA as described in table 3.21 and written to the result vector.

```
  tmp = VectorLoad(op0);
  result = TextureSample(tmp.x, tmp.y, tmp.z, 0.0, op1, op2);
```

Add Section 3.11.X, Fragment Program Flow Control Instruction Set
(immediately after Section 3.11.6, Fragment Program Texture Instruction Set)

3.11.X.1, BRK: Break

The BRK instruction conditionally transfers control to the instruction immediately following the next ENDLOOP or ENDREP instruction. A BRK instruction has no effect if the condition code test evaluates to FALSE.

The following pseudocode describes the operation of the instruction:

```
  if (TestCC(cc.c***) || TestCC(cc.*c**) ||
      TestCC(cc.**c*) || TestCC(cc.***c)) {
    continue execution at instruction following the next ENDLOOP or
      ENDREP;
  }
```

3.11.X.2, CAL: Subroutine Call

The CAL instruction conditionally transfers control to the instruction following the label specified in the instruction. A CAL instruction has no effect if the condition code test evaluates to FALSE.

When executed, the CAL instruction pushes a reference to the instruction immediately following the CAL instruction onto the call stack. When a matching RET instruction is executed, execution will continue at that instruction after executing the matching RET instruction.

Implementations may have a limited call stack. If the number of CAL instructions that have been performed without returning is MAX_PROGRAM_CALL_DEPTH_NV, a CAL instruction will cause the call stack to overflow and the fragment program to terminate.

The following pseudocode describes the operation of the instruction:

```
  if (TestCC(cc.c***) || TestCC(cc.*c**) ||
      TestCC(cc.**c*) || TestCC(cc.***c)) {

    // Check for call stack overflow.
    if (callStackDepth >= MAX_PROGRAM_CALL_DEPTH_NV) {
      terminate fragment program;
    }

    push instruction following the CAL instruction on the call stack;
    continue execution at instruction following <branchLabel>;
  }
```

3.11.X.3, ELSE: Beginning of ELSE Block

The ELSE instruction signifies the end of the "execute if true" portion of an IF/ELSE/ENDIF block.

If the condition evaluated at the IF statement was TRUE, when a program reaches the ELSE statement, it has completed the entire "execute if true" portion of the IF/ELSE/ENDIF block. Execution will continue at the corresponding ENDIF instruction.

If the condition evaluated at the IF statement was FALSE, program execution would skip over the entire "execute if true" portion of the IF/ELSE/ENDIF block, including the ELSE instruction.

3.11.X.4, ENDIF: End of IF/ELSE Block

The ENDIF instruction signifies the end of an IF/ELSE/ENDIF block. It has no other effect on program execution.

3.11.X.5, ENDLOOP: End of LOOP Block

The ENDLOOP instruction specifies the end of a LOOP block. When an ENDLOOP instruction executes, the loop count is decremented and the loop index increment value is added to the loop index (A0.x). If the decremented loop count is greater than zero, execution continues at the top of the LOOP block.

```
  LoopCount--;
  LoopIndex += LoopIncr;
  if (LoopCount > 0) {
    continue execution at instruction following corresponding LOOP
      instruction;
  }
```

3.11.X.6, ENDREP: End of REP Block

The ENDREP instruction specifies the end of a REP block. When an ENDREP instruction executes, the loop count is decremented. If the decremented loop count is greater than zero, execution continues at the top of the REP block.

```
  LoopCount--;
  if (LoopCount > 0) {
    continue execution at instruction following corresponding LOOP
      instruction;
  }
```

3.11.X.7, IF: Beginning of IF Block

The IF instruction conditionally transfers control to the instruction immediately following the corresponding ELSE instruction (if present) or ENDIF instruction (if no ELSE is present).

Implementations may have a limited ability to nest IF blocks at run time. If the number of IF/ENDIF blocks that are currently active is MAX_PROGRAM_IF_DEPTH_NV, an IF instruction will cause the fragment program to terminate. If an IF instruction is executed inside a subroutine, any active IF/ENDIF blocks in the calling code count against this limit.

```
  if (IF block nested too deeply) {
    terminate fragment program;
  }

  // Evaluate the condition.  If the condition is true, continue at the
  // next instruction.  Otherwise, continue at the
  if (TestCC(cc.c***) || TestCC(cc.*c**) ||
      TestCC(cc.**c*) || TestCC(cc.***c)) {
    continue execution at the next instruction;
  } else if (IF block contains an ELSE statement) {
    continue execution at instruction following corresponding ELSE;
  } else {
    continue execution at instruction following corresponding ENDIF;
  }
```

3.11.X.8, LOOP: Beginning of LOOP Block

The LOOP instruction begins a LOOP block. The x, y, and z components of the single vector operand specify the initial values for the loop count, loop index, and loop index increment, respectively.

The loop count indicates the number of times the instructions between the LOOP and corresponding ENDLOOP instruction will be executed. If the initial value of the loop count is not positive, the entire block is skipped and execution continues at the corresponding ENDLOOP instruction.

The loop index (A0.x) can be used for indirect addressing in the set of texture coordinate fragment attributes. A fragment program can only use the loop index of the current LOOP block; loop indices for containing LOOP blocks are not available.

Implementations may have a limited ability to nest LOOP and REP blocks at run time. If the number of LOOP/ENDLOOP and REP/ENDREP blocks that have not completed is MAX_PROGRAM_LOOP_DEPTH_NV, a LOOP instruction will cause the fragment program to terminate. If a LOOP instruction is executed inside a subroutine, any active LOOP/ENDLOOP or REP/ENDREP blocks in the calling code count against this limit.

```
  if (LOOP block nested too deeply) {
    terminate fragment program;
  }

  // Set up loop information for the new nesting level.
  tmp = VectorLoad(op0);
  LoopCount = floor(op0.x);
  LoopIndex = floor(op0.y);
  LoopIncr  = floor(op0.z);
  if (LoopCount <= 0) {
    continue execution at the corresponding ENDLOOP;
  }
```

LOOP blocks do not support fully general branching -- a fragment program will fail to load if the vector operand is not a program parameter.

3.11.X.9, REP: Beginning of REP Block

The REP instruction begins a REP block. The x component of the single vector operand specifies the initial value for the loop count. REP blocks are completely identical to LOOP blocks except that they don't use the loop index at all.

The loop count indicates the number of times the instructions between the REP and corresponding ENDREP instruction will be executed. If the initial value of the loop count is not positive, the entire block is skipped and execution continues at the instruction following the corresponding ENDREP instruction.

Implementations may have a limited ability to nest LOOP and REP blocks at run time. If the number of LOOP/ENDLOOP and REP/ENDREP blocks that have not completed is MAX_PROGRAM_LOOP_DEPTH_NV, a REP instruction will cause the fragment program to terminate. If a REP instruction is executed inside a subroutine, any active LOOP/ENDLOOP or REP/ENDREP blocks in the calling code count against this limit.

```
  if (REP block nested too deeply) {
    terminate fragment program;
  }

  // Set up loop information for the new nesting level.
  tmp = VectorLoad(op0);
  LoopCount = floor(op0.x);
  if (LoopCount <= 0) {
    continue execution at the corresponding ENDREP;
  }
```

REP blocks do not support fully general branching -- a fragment program will fail to load if the vector operand is not a program parameter.

3.11.X.10, RET: Subroutine Return

The RET instruction conditionally returns from a subroutine initiated by a CAL instruction. A RET instruction has no effect if the condition code test evaluates to FALSE.

When executed, the RET instruction pops a reference to the instruction immediately following the corresponding CAL instruction onto the call stack and continues execution at that instruction.

If a RET instruction is issued when the call stack is empty, the fragment program is terminated.

```
  if (TestCC(cc.c***) || TestCC(cc.*c**) ||
      TestCC(cc.**c*) || TestCC(cc.***c)) {

    if (callStackDepth <= 0) {
      terminate fragment program;
    }

    pop instruction following the CAL instruction off the call stack;
    continue execution at that instruction;
  }
```

Additions to Chapter 4 of the OpenGL 1.4 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 1.4 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 1.4 Specification (State and State Requests)

None.

Additions to Appendix A of the OpenGL 1.4 Specification (Invariance)

None.

Additions to the AGL/GLX/WGL Specifications

None.

Dependencies on ARB_fragment_program

ARB_fragment_program is required.

This specification and NV_fragment_program_option are based on a modified version of the grammar published in the ARB_fragment_program specification. This modified grammar includes a few structural changes to better accommodate new functionality from this and other extensions, but should be functionally equivalent to the ARB_fragment_program grammar. See NV_fragment_program_option for details on the base grammar.

Dependencies on NV_fragment_program2_option

NV_fragment_program_option is required.

If the NV_fragment_program2 program option is specified, all the functionality described in both this extension and the NV_fragment_program_option specification is available.

GLX Protocol

None.

Errors

None.

New State

None.

New Implementation Dependent State

Get Value	Type	Get Command	Min Value	Description	Sec	Attrib
MAX_PROGRAM_EXEC_INSTRUCTIONS_NV	Z+	GetProgramivARB	65536	maximum program execution inst- ruction count	3.11.4.X	-
MAX_PROGRAM_CALL_DEPTH_NV	Z+	GetProgramivARB	4	maximum program call stack depth	3.11.4.X	-
MAX_PROGRAM_IF_DEPTH_NV	Z+	GetProgramivARB	48	maximum program if nesting	3.11.4.X	-
MAX_PROGRAM_LOOP_DEPTH_NV	Z+	GetProgramivARB	4	maximum program loop nesting	3.11.4.X	-
MAX_PROGRAM_LOOP_COUNT_NV	Z+	GetProgramivARB	255	maximum program initial loop count	3.11.4.X	-

(add to Table X.10. New Implementation-Dependent Values Introduced by ARB_fragment_program. Values queried by GetProgramivARB require a <pname> of FRAGMENT_PROGRAM_ARB.)

Revision History

```
    Rev.  Date      Author   Changes
    ----  --------  -------  ------------------------------------------
    8     08/04/04  pbrown   Fixed two typos in the TXL instruction.

    7     07/08/04  pbrown   Fixed entries for KIL and RFL in the opcode
                             table.

    6     05/16/04  pbrown   Documented that "A0" is a pre-defined address
                             register variable for the purposes of the
                             grammar, and that no other address register
                             variables can be declared.

    5     --------  pbrown   Internal pre-release revisions.
```

Name

NV_fragment_program4

Name Strings

(none)

Contact

Pat Brown, NVIDIA Corporation (pbrown 'at' nvidia.com)

Status

Shipping for GeForce 8 Series (November 2006)

Version

```
Last Modified Date:         11/06/2007
NVIDIA Revision:            4
```

Number

335

Dependencies

OpenGL 1.1 is required.

NV_gpu_program4 is required. This extension is supported if "GL_NV_gpu_program4" is found in the extension string.

ATI_draw_buffers and ARB_draw_buffers trivially affects the definition of this specification.

ARB_fragment_program_shadow trivially affects the definition of this specification.

NV_primitive_restart trivially affects the definition of this extension.

This extension is written against the OpenGL 2.0 specification.

Overview

This extension builds on the common assembly instruction set infrastructure provided by NV_gpu_program4, adding fragment program-specific features.

This extension provides interpolation modifiers to fragment program attributes allowing programs to specify that specified attributes be flat-shaded (constant over a primitive), centroid-sampled (multisample rendering), or interpolated linearly in screen space. The set of input and output bindings provided includes all bindings supported by ARB_fragment_program. Additional input bindings are provided to determine whether fragments were generated by front- or back-facing primitives ("fragment.facing"), to identify the individual primitive used to generate the fragment ("primitive.id"), and to determine distances to user clip planes ("fragment.clip[n]"). Additionally generic input attributes allow a fragment program to receive a greater number of attributes from previous pipeline stages than possible using only the pre-defined fixed-function attributes.

By and large, programs written to ARB_fragment_program can be ported directly by simply changing the program header from "!!ARBfp1.0" to "!!NVfp4.0", and then modifying instructions to take advantage of the expanded feature set. There are a small number of areas where this extension is not a functional superset of previous fragment program extensions, which are documented in the NV_gpu_program4 specification.

New Procedures and Functions

None.

New Tokens

None.

Additions to Chapter 2 of the OpenGL 2.0 Specification (OpenGL Operation)

Modify Section 2.X, GPU Programs

(insert after second paragraph)

Fragment Programs

Fragment programs are used to compute the transformed attributes of a fragment, in lieu of the set of fixed-function operations described in sections 3.8 through 3.10. Fragment programs are run on a single fragment at a time, and the state of neighboring fragments is not explicitly available. (In practice, fragment programs may be run on a block of fragments, and neighboring fragments' attributes may be used for texture LOD calculations or partial derivative approximation.) The inputs available to a fragment program are the interpolated attributes of a fragment, which include (among other things) window-space position, primary and secondary colors, and texture coordinates. The results of the program are one (or more) colors and possibly a new window Z coordinate. A fragment program can not modify the (X,Y) location of the fragment.

Modify Section 2.X.2, Program Grammar

(replace third paragraph)

Fragment programs are required to begin with the header string "!!NVfp4.0". This header string identifies the subsequent program body as being a fragment program and indicates that it should be parsed according to the base NV_gpu_program4 grammar plus the additions below. Program string parsing begins with the character immediately following the header string.

(add the following grammar rules to the NV_gpu_program4 base grammar)

```
<instruction>           ::= <SpecialInstruction>

<varModifier>           ::= <interpModifier>

<SpecialInstruction>    ::= "KIL" <opModifiers> <killCond>
                          | "DDX" <opModifiers> <instResult> ","
                            <instOperandV>
                          | "DDY" <opModifiers> <instResult> ","
                            <instOperandV>

<killCond>              ::= <instOperandV>

<interpModifier>        ::= "FLAT"
                          | "CENTROID"
                          | "NOPERSPECTIVE"

<attribBasic>           ::= <fragPrefix> "fogcoord"
                          | <fragPrefix> "position"
                          | <fragPrefix> "facing"
                          | <attribTexCoord> <optArrayMemAbs>
                          | <attribClip> <arrayMemAbs>
                          | <attribGeneric> <arrayMemAbs>
                          | "primitive" "." "id"

<attribColor>           ::= <fragPrefix> "color"

<attribMulti>           ::= <attribTexCoord> <arrayRange>
                          | <attribClip> <arrayRange>
                          | <attribGeneric> <arrayRange>

<attribTexCoord>        ::= <fragPrefix> "texcoord"

<attribClip>            ::= <fragPrefix> "clip"

<attribGeneric>         ::= <fragPrefix> "attrib"

<fragPrefix>            ::= "fragment" "."

<resultBasic>           ::= <resPrefix> "color" <resultOptColorNum>
                          | <resPrefix> "depth"

<resultOptColorNum>     ::= /* empty */

<resPrefix>             ::= "result" "."
```

(add the following subsection to section 2.X.3.1, Program Variable Types)

Explicitly declared fragment program attribute variables may have one or more interpolation modifiers that control how per-fragment values are computed.

An attribute variable declared as "FLAT" will be flat-shaded. For such variables, the value of the attribute will be constant over an entire primitive and will taken from the provoking vertex of the primitive, as described in Section 2.14.7. If "FLAT" is not specified, attributes will be interpolated as described in Chapter 3, with the exception that attribute variables bound to colors will still be flat-shaded if the shade model (section 2.14.7) is FLAT. If an attribute variable declared as "FLAT" corresponds to a texture coordinate replaced by a point sprite (s,t) value (section 3.3.1), the value of the attribute is undefined.

An attribute variable declared as "CENTROID" will be interpolated using a point on or inside the primitive, if possible, when doing multisample line or polygon rasterization (sections 3.4.4 and 3.5.6). This method can avoid artifacts during multisample rasterization when some samples of a pixel are covered, but the sample location used is outside the primitive. Note that when centroid sampling, the sample points used to generate attribute values for adjacent pixels may not be evenly spaced, which can lead to artifacts when evaluating partial derivatives or performing texture LOD calculations needed for mipmapping. If "CENTROID" is not specified, attributes may be sampled anywhere inside the pixel as permitted by the specification, including at points outside the primitive.

An attribute variable declared as "NOPERSPECTIVE" will be interpolated using a method that is linear in screen space, as described in equation 3.7 and the appoximation that follows equation 3.8. If "NOPERSPECTIVE" is not specified, attributes must be interpolated with perspective correction, as described in equations 3.6 and 3.8. When clipping lines or polygons, an alternate method is used to compute the attributes of vertices introduced by clipping when they are specified as "NOPERSPECTIVE" (section 2.14.8).

Implicitly declared attribute variables (bindings used directly in a program instruction) will inherit the interpolation modifiers of any explicitly declared attribute variable using the same binding. If no such variable exists, default interpolation modes will be used.

For fragments generated by point primitives, DrawPixels, and Bitmap, interpolation modifiers have no effect.

Implementations are not required to support arithmetic interpolation of integer values written by a previous pipeline stage. Integer fragment program attribute variables must be flat-shaded; a program will fail to load if it declares a variable with the "INT" or "UINT" data type modifiers without the "FLAT" interpolation modifier.

There are several additional limitations on the use of interpolation modifiers. A fragment program will fail to load:

* if an interpolation modifier is specified when declaring a non-attribute variable,

* if the same interpolation modifier is specified more than once in a single declaration (e.g., "CENTROID CENTROID ATTRIB"),

* if the "FLAT" modifier is used together with either "CENTROID" or "NOPERSPECTIVE" in a single declaration,

* if any interpolation modifier is specified when declaring a variable bound to a fragment's position, face direction, fog coordinate, or any interpolated clip distance,

* if multiple attribute variables with different interpolation modifiers are bound to the same fragment attribute, or

* if one variable is bound to the fragment's primary color and a second variable with different interpolation modifiers is bound the the fragment's secondary color.

(add the following subsection to section 2.X.3.2, Program Attribute Variables)

Fragment program attribute variables describe the attributes of a fragment produced during rasterization. The set of available bindings is enumerated in Table X.X.

Most attributes correspond to per-vertex attributes that are interpolated over a primitive; such attributes are subject to the interpolation modifiers described in section 2.X.3.1. The fragment's position, facing, and primitive IDs are the exceptions, and are generated specially during rasterization. Since two-sided color selection occurs prior to rasterization, there are no distinct "front" or "back" colors available to fragment programs. A single set of colors is available, which corresponds to interpolated front or back vertex colors.

If geometry programs are enabled, attributes will be obtained by interpolating per-vertex outputs written by the geometry program. If geometry programs are disabled, but vertex programs are enabled, attributes will be obtained by interpolating per-vertex outputs written by the vertex program. In either case, the fragment program attributes should be read using the same component data type used to write the vertex output attributes in the geometry or vertex program. The value of any attribute corresponding to a vertex output not written by the geometry or vertex program is undefined.

If neither geometry nor vertex programs are used, attributes will be obtained by interpolating per-vertex values computed by fixed-function vertex processing. All interpolated fragment attributes should be read as floating-point values.

Fragment Attribute Binding	Components	Underlying State
fragment.color	(r,g,b,a)	primary color
fragment.color.primary	(r,g,b,a)	primary color
fragment.color.secondary	(r,g,b,a)	secondary color
fragment.texcoord	(s,t,r,q)	texture coordinate, unit 0
fragment.texcoord[n]	(s,t,r,q)	texture coordinate, unit n
fragment.fogcoord	(f,-,-,-)	fog distance/coordinate
* fragment.clip[n]	(c,-,-,-)	interpolated clip distance n
fragment.attrib[n]	(x,y,z,w)	generic interpolant n
fragment.texcoord[n..o]	(s,t,r,q)	texture coordinates n thru o
* fragment.clip[n..o]	(c,-,-,-)	clip distances n thru o
fragment.attrib[n..o]	(x,y,z,w)	generic interpolants n thru o
* fragment.position	(x,y,z,1/w)	window position
* fragment.facing	(f,-,-,-)	fragment facing
* primitive.id	(id,-,-,-)	primitive number

Table X.X: Fragment Attribute Bindings. The "Components" column indicates the mapping of the state in the "Underlying State" column. Bindings containing "[n]" require an integer value of <n> to select an individual item. Interpolation modifiers are not supported on variables that use bindings labeled with "*".

If a fragment attribute binding matches "fragment.color" or "fragment.color.primary", the "x", "y", "z", and "w" components of the fragment attribute variable are filled with the "r", "g", "b", and "a" components, respectively, of the fragment's primary color.

If a fragment attribute binding matches "fragment.color.secondary", the "x", "y", "z", and "w" components of the fragment attribute variable are filled with the "r", "g", "b", and "a" components, respectively, of the fragment's secondary color.

If a fragment attribute binding matches "fragment.texcoord" or "fragment.texcoord[n]", the "x", "y", "z", and "w" components of the fragment attribute variable are filled with the "s", "t", "r", and "q" components, respectively, of the fragment texture coordinates for texture unit <n>. If "[n]" is omitted, texture unit zero is used.

If a fragment attribute binding matches "fragment.fogcoord", the "x" component of the fragment attribute variable is filled with either the fragment eye distance or the fog coordinate, depending on whether the fog source is set to FRAGMENT_DEPTH_EXT or FOG_COORDINATE_EXT, respectively. The "y", "z", and "w" coordinates are undefined.

If a fragment attribute binding matches "fragment.clip[n]", the "x" component of the fragment attribute variable is filled with the interpolated value of clip distance <n>, as written by the vertex or geometry program. The "y", "z", and "w" components of the variable are undefined. If fixed-function vertex processing or position-invariant vertex programs are used with geometry programs disabled, clip distances are obtained by interpolating the per-clip plane dot product:

$(p_1'\ p_2'\ p_3'\ p_4')$ dot $(x_e\ y_e\ z_e\ w_e)$,

for clip plane <n> as described in section 2.12. The clip distance for clip plane <n> is undefined if clip plane <n> is disabled.

If a fragment attribute binding matches "fragment.attrib[n]", the "x", "y", "z", and "w" components of the fragment attribute variable are filled with the "x", "y", "z", and "w" components of generic interpolant <n>. All generic interpolants will be undefined when used with fixed-function vertex processing with no geometry program enabled.

If a fragment attribute binding matches "fragment.texcoord[n..o]", "fragment.clip[n..o]", or "fragment.attrib[n..o]", a sequence of 1+<o>-<n> bindings is created. For texture coordinate bindings, it is as though the sequence "fragment.texcoord[n], fragment.texcoord[n+1], ... fragment.texcoord[o]" were specfied. These bindings are available only in explicit declarations of array variables. A program will fail to load if <n> is greater than <o>.

If a fragment attribute binding matches "fragment.position", the "x" and "y" components of the fragment attribute variable are filled with the floating-point (x,y) window coordinates of the fragment center, relative to the lower left corner of the window. The "z" component is filled with the fragment's z window coordinate. If z window coordinates are represented internally by the GL as fixed-point values, the z window coordinate undergoes an implied conversion to floating point. This conversion must leave the values 0 and 1 invariant. The "w" component is filled with the reciprocal of the fragment's clip w coordinate.

If a fragment attribute binding matches "fragment.facing", the "x" component of the fragment attribute variable is filled with +1.0 or -1.0, depending on the orientation of the primitive producing the fragment. If the fragment is generated by a back-facing polygon (including point- and line-mode polygons), the facing is -1.0; otherwise, the facing is +1.0. The "y", "z", and "w" coordinates are undefined.

If a fragment attribute binding matches "primitive.id", the "x" component of the fragment attribute variable is filled with a single integer. If a geometry program is active, this value is obtained by taking the primitive ID value emitted by the geometry program for the provoking vertex. If no geometry program is active, the value is the number of primitives processed by the rasterizer since the last time Begin was called (directly or indirectly via vertex array functions). The first primitive generated after a Begin is numbered zero, and the primitive ID counter is incremented after every individual point, line, or polygon primitive is processed. For polygons drawn in point or line mode, the primitive ID counter is incremented only once, even though multiple points or lines may be drawn. For QUADS and QUAD_STRIP primitives that are decomposed into triangles, the primitive ID is incremented after each complete quad is processed. For POLYGON primitives, the primitive ID counter is zero. The primitive ID is zero for fragments generated by DrawPixels or Bitmap. Restarting a primitive topology using the primitive restart index has no effect on the primitive ID counter. The "y", "z", and "w" components of the variable are always undefined.

(add the following subsection to section 2.X.3.5, Program Results.)

Fragment programs produce final fragment values, and the set of result variables available to such programs correspond to the final attributes of a fragment. Fragment program result variables may not be declared as arrays.

The set of allowable result variable bindings is given in Table X.X.

Binding	Components	Description
result.color	(r,g,b,a)	color
result.color[n]	(r,g,b,a)	color output n
result.depth	(*,*,d,*)	depth coordinate

Table X.X: Fragment Result Variable Bindings.
Components labeled "*" are unused.

If a result variable binding matches "result.color", updates to the "x", "y", "z", and "w" components of the result variable modify the "r", "g", "b", and "a" components, respectively, of the fragment's output color.

If a result variable binding matches "result.color[n]" and the ARB_draw_buffers program option is specified, updates to the "x", "y", "z", and "w" components of the color result variable modify the "r", "g", "b", and "a" components, respectively, of the fragment output color numbered <n>. If the ARB_draw_buffers program option is not specified, the "result.color[n]" binding is unavailable.

If a result variable binding matches "result.depth", updates to the "z" component of the result variable modify the fragment's output depth value. If the "result.depth" binding is not in used in a variable written to by any instruction in the fragment program, the interpolated depth value produced by rasterization is used as if fragment program mode is not enabled. Otherwise, the value written by the fragment program is used, and the fragment's final depth value is undefined if the program did not end up writing a depth value due to flow control or write masks. Writes to any component of depth other than the "z" component have no effect.

(modify Table X.13 in section 2.X.4, Program Instructions, to include the following.)

	Modifiers								
Instruction	F	I	C	S	H	D	Inputs	Out	Description
DDX	X	-	X	X	X	F	v	v	partial derivative relative to X
DDY	X	-	X	X	X	F	v	v	partial derivative relative to Y
KIL	X	X	-	-	X	F	vc	-	kill fragment

(add the following subsection to section 2.X.5, Program Options.)

Section 2.X.5.Y, Fragment Program Options

+ Fixed-Function Fog Emulation (ARB_fog_exp, ARB_fog_exp2, ARB_fog_linear)

If a fragment program specifies one of the options "ARB_fog_exp", "ARB_fog_exp2", or "ARB_fog_linear", the program will apply fog to the program's final color using a fog mode of EXP, EXP2, or LINEAR, respectively, as described in section 3.10.

When a fog option is specified in a fragment program, semantic restrictions are added to indicate that a fragment program will fail to load if the number of temporaries it contains exceeds the implementation-dependent limit minus 1, if the number of attributes it contains exceeds the implementation-dependent limit minus 1, or if the number of parameters it contains exceeds the implementation-dependent limit minus 2.

Additionally, when the ARB_fog_exp option is specified in a fragment program, a semantic restriction is added to indicate that a fragment program will fail to load if the number of instructions or ALU instructions it contains exceeds the implementation-dependent limit minus 3. When the ARB_fog_exp2 option is specified in a fragment program, a semantic restriction is added to indicate that a fragment program will fail to load if the number of instructions or ALU instructions it contains exceeds the implementation-dependent limit minus 4. When the ARB_fog_linear option is specified in a fragment program, a semantic restriction is added to indicate that a fragment program will fail to load if the number of instructions or ALU instructions it contains exceeds the implementation-dependent limit minus 2.

Only one fog application option may be specified by any given fragment program. A fragment program that specifies more than one of the program options "ARB_fog_exp", "ARB_fog_exp2", and "ARB_fog_linear", will fail to load.

+ Precision Hints (ARB_precision_hint_fastest, ARB_precision_hint_nicest)

Fragment program computations are carried out at an implementation-dependent precision. However, some implementations may be able to perform fragment program computations at more than one precision, and may be able to trade off computation precision for performance.

If a fragment program specifies the "ARB_precision_hint_fastest" program option, implementations should select precision to minimize program execution time, with possibly reduced precision. If a fragment program specifies the "ARB_precision_hint_nicest" program option, implementations should maximize the precision, with possibly increased execution time.

Only one precision control option may be specified by any given fragment program. A fragment program that specifies both the "ARB_precision_hint_fastest" and "ARB_precision_hint_nicest" program options will fail to load.

+ Multiple Color Outputs (ARB_draw_buffers, ATI_draw_buffers)

If a fragment program specifies the "ARB_draw_buffers" or "ATI_draw_buffers" option, it will generate multiple output colors, and the result binding "result.color[n]" is allowed, as described in section 2.X.3.5. If this option is not specified, a fragment program that attempts to bind "result.color[n]" will fail to load, and only "result.color" will be allowed.

The multiple color outputs will typically be written to an ordered list of draw buffers in the manner described in the ARB_draw_buffers extension specification.

+ Fragment Program Shadows (ARB_fragment_program_shadow)

The ARB_fragment_program_shadow option introduced a set of "SHADOW" texture targets and made the results of depth texture lookups undefined unless the texture format and compare mode were consistent with the target provided in the fragment program instruction. This behavior is enabled by default in NV_gpu_program4; specifying the option is not illegal but has no additional effect.

(add the following subsection to section 2.X.8, Program Instruction Set.)

Section 2.X.8.Z, DDX: Partial Derivative Relative to X

The DDX instruction computes approximate partial derivatives of the four components of the single floating-point vector operand with respect to the X window coordinate to yield a result vector. The partial derivatives are evaluated at the sample location of the pixel.

```
  f = VectorLoad(op0);
  result = ComputePartialX(f);
```

Note that the partial derivates obtained by this instruction are approximate, and derivative-of-derivate instruction sequences may not yield accurate second derivatives. Note also that the sample locations for attributes declared with the CENTROID interpolation modifier may not be evenly spaced, which can lead to artifacts in derivative calculations.

DDX supports only floating-point data type modifiers and is available only to fragment programs.

Section 2.X.8.Z, DDY: Partial Derivative Relative to Y

The DDY instruction computes approximate partial derivatives of the four components of the single operand with respect to the Y window coordinate to yield a result vector. The partial derivatives are evaluated at the center of the pixel.

```
  f = VectorLoad(op0);
  result = ComputePartialY(f);
```

Note that the partial derivates obtained by this instruction are approximate, and derivative-of-derivate instruction sequences may not yield accurate second derivatives. Note also that the sample locations for attributes declared with the CENTROID interpolation modifier may not be evenly spaced, which can lead to artifacts in derivative calculations.

DDY supports only floating-point data type modifiers and is available only to fragment programs.

Section 2.X.8.Z, KIL: Kill Fragment

The KIL instruction evaluates a condition and kills a fragment if the test passes. A fragment killed by the KIL instruction is discarded, and will not be seen by subsequent stages of the pipeline.

A KIL instruction may be specified using either a floating-point vector operand or a condition code test.

If a floating-point vector is provided, the fragment is discarded if any of its components are negative:

```
  tmp = VectorLoad(op0);
  if ((tmp.x < 0) || (tmp.y < 0) ||
      (tmp.z < 0) || (tmp.w < 0))
  {
      exit;
  }
```

If a condition code test is provided, the fragment is discarded if any component of the test passes:

```
  if (TestCC(rc.c***) || TestCC(rc.*c**) ||
      TestCC(rc.**c*) || TestCC(rc.***c))
  {
     exit;
  }
```

KIL supports no data type modifiers. If a vector operand is provided, it must have floating-point components.

KIL is available only to fragment programs.

Replace Section 2.14.8, and rename it to "Vertex Attribute Clipping" (p. 70).

After lighting, clamping or masking and possible flatshading, vertex attributes, including colors, texture and fog coordinates, shader varying variables, and point sizes computed on a per vertex basis, are clipped. Those attributes associated with a vertex that lies within the clip volume are unaffected by clipping. If a primitive is clipped, however, the attributes assigned to vertices produced by clipping are produced by interpolating attributes along the clipped edge.

Let the attributes assigned to the two vertices P_1 and P_2 of an unclipped edge be a_1 and a_2. The value of t (section 2.12) for a clipped point P is used to obtain the attribute associated with P as

```
  a = t * a_1 + (1-t) * a_2
```

unless the attribute is specified to be interpolated without perspective correction in a fragment program. In that case, the attribute associated with P is

```
  a = t' * a_1 + (1-t') * a_2
```

where

```
  t' = (t * w_1) / (t * w_1 + (1-t) * w_2)
```

and w_1 and w_2 are the w clip coordinates of P_1 and P_2, respectively. If w_1 or w_2 is either zero or negative, the value of the associated attribute is undefined.

Additions to Chapter 3 of the OpenGL 2.0 Specification (Rasterization)

    None

Additions to Chapter 4 of the OpenGL 2.0 Specification (Per-Fragment Operations and the Frame Buffer)

    None

Additions to Chapter 5 of the OpenGL 2.0 Specification (Special Functions)

    None

Additions to Chapter 6 of the OpenGL 2.0 Specification (State and State Requests)

    None

Additions to the AGL/GLX/WGL Specifications

    None

Dependencies on ARB_draw_buffers and ATI_draw_buffers

If neither ARB_draw_buffers nor ATI_draw_buffers is supported, then the discussion of the ARB_draw_buffers option in section 2.X.5.Y should be removed, as well as the result bindings of the form "result.color[n]" and "result.color[n..o]".

Dependencies on ARB_fragment_program_shadow

If ARB_fragment_program_shadow is not supported, then the discussion of the ARB_fragment_program_shadow option in section 2.X.5.Y should be removed.

Dependencies on NV_primitive_restart

The spec describes the behavior that primitive restart does not affect the primitive ID counter, including for POLYGON primitives (where one could argue that the restart index starts a new primitive without a new Begin to reset the count. If NV_primitive_restart is not supported, references to that extension in the discussion of the "primitive.id" attribute should be removed.

Errors

    None

New State

    None

New Implementation Dependent State

    None

Issues

(1) How should special interpolation controls be specified?

RESOLVED: As a special modifier to fragment program attribute variable declarations. It was decided that the fragment program was the most natural place to put the control. This wouldn't require making a large number of related state changes controlling interpolation whenever the fragment program used. The final mechanism using special interpolation modifiers was chosen because it fit well with the other variable modifiers (for data storage size and data type) provided by NV_gpu_program4. Examples:

```
FLAT ATTRIB texcoords[4] = { fragment.texcoord[0..3] };
CENTROID ATTRIB texcoord4 = fragment.texcoord[4];
CENTROID NOPERSPECTIVE ATTRIB
  attribs[3] = { fragment.attrib[0..2] };
```

There were a variety of options considered, including:

* special declarations in vertex or geometry programs to specify the interpolation type,

* special declarations in the fragment program to specify one or more interpolation type modifiers per binding, such as:

```
INTERPOLATE fragment.texcoord[0..3], FLAT;
INTERPOLATE fragment.texcoord[4], CENTROID;
INTERPOLATE fragment.attrib[0..2], CENTROID, NOPERSPECTIVE;
```

* fixed-function state specifying the interpolation mode

```
glInterpolateAttribNV(GL_TEXTURE0, GL_FLAT);
glInterpolateAttribNV(GL_GENERIC_ATTRIB0, GL_CENTROID_NV);
```

Recent updates to GLSL provide similar functionality (for centroid) with a similar approach, using a modifier on varying variable declarations.

(2) How should perspective-incorrect interpolation (linear in screen space) and clipping interact?

RESOLVED: Primitives with attributes specified to be perspective-incorrect should be clipped so that the vertices introduced by clipping should have attribute values consistent with the interpolation mode. We do not want to have large color shifts introduced by clipping a perspective-incorrect attribute. For example, a primitive that approaches, but doesn't cross, a frustum clip plane should look pretty much identical to a similar primitive that just barely crosses the clip plane.

Clipping perspective-incorrect interpolants that cross the W==0 plane is very challenging. The attribute clipping equation provided in the spec effectively projects all the original vertices to screen space while ignoring the X and Y frustum clip plane. As W approaches zero, the projected X/Y window coordinates become extremely large. When clipping an edge with one vertex inside the frustum and the other out near infinity (after projection, due to W approaching zero), the interpolated attribute for the entire visible portion of the edge should almost exactly match the attribute value of the visible vertex.

If an outlying vertex approaches and then goes past W==0, it can be said to go "to infinity and beyond" in screen space. The correct answer for screen-linear interpolation is no longer obvious, at least to the author of this specification. Rather than trying to figure out what the "right" answer is or if one even exists, the results of clipping such edges is specified as undefined.

(3) If a shader wants to use interpolation modifiers without using declared variables, is that possible?

RESOLVED: Yes. If "dummy" variables are declared, all interpolants bound to that variable will get the variable's interpolation modifiers. In the following program:

```
FLAT ATTRIB tc02[3] = { fragment.texcoord[0..2] };
MOV R0, fragment.texcoord[1];
MOV R1, fragment.texcoord[3];
```

The variable R0 will get texture coordinate set 1, which will be flat-shaded due to the declaration of "tc02". The variable R1 will get texture coordinate set 3, which will be smooth shaded (default).

(4) Is it possible to read the same attribute with different interpolation modifiers?

RESOLVED: No. A program that tries to do that will fail to compile.

(5) Why can't fragment program results be declared as arrays?

RESOLVED: This is a limitation of the programming model. If an implementation needs to do run-time indexing of fragment program result variables (effectively writing to "result.color[A0.x]"), code such as the following can be used:

```
TEMP colors[4];
...
MOV colors[A0.x], R1;
MOV colors[3], 12.3;
...
# end of the program
MOV result.color[0], colors[0];
MOV result.color[1], colors[1];
MOV result.color[2], colors[2];
MOV result.color[3], colors[3];
```

(6) Do clip distances require that the corresponding clip planes be enabled to be read by a fragment program?

RESOLVED: No.

(7) How do primitive IDs work with fragment programs?

RESOLVED: If a geometry program is enabled, the primitive ID is consumed by the geometry program and is not automatically available to the fragment program. If the fragment program needs a primitive ID in this case, the geometry program can write out a primitive ID using the "result.primid" binding, and the fragment program will see the primitive ID written for the provoking vertex.

If no geometry program is enabled, the primitive ID is automatically available, and specifies the number of primitives (points, lines, or triangles) processed by since the last explicit or implicit Begin call.

(8) What is the primitive ID for non-geometry commands that generate fragments, such as DrawPixels, Bitmap, and CopyPixels.

RESOLVED: Zero.

(9) How does the FLAT interpolation modifier interact with point sprite coordinate replacement?

RESOLVED: The value of such attributes are undefined. Specifying these two operations together is self-contradictory -- FLAT asks for an interpolant that is constant over a primitive, and point sprite coordinate interpolation asks for an interpolant that is non-constant over a point sprite.

Revision History

```
    Rev.    Date     Author    Changes
    ----  --------  --------  -----------------------------------------
     4    11/06/07  pbrown    Documented interaction between the FLAT
                              interpolation modifier and point sprite
                              coordinate replacement.

    1-3             pbrown    Internal spec development.
```

Name

    NV_framebuffer_multisample_coverage

Name Strings

    GL_NV_framebuffer_multisample_coverage

Contact

    Mike Strauss, NVIDIA Corporation (mstrauss 'at' nvidia.com)

Status

    Shipping in NVIDIA Release 95 drivers (November 2006)

    Functionaltiy supported by GeForce 8800

Version

    Last Modified Date: November 6, 2006
    Revision #5

Number

    336

Dependencies

    Requires GL_EXT_framebuffer_object.

    Requires GL_EXT_framebuffer_blit.

    Requires GL_EXT_framebuffer_multisample.

    Written based on the wording of the OpenGL 1.5 specification.

Overview

    This extension extends the EXT_framebuffer_multisample
    specification by providing a new function,
    RenderBufferStorageMultisampleCoverageNV, that distinguishes
    between color samples and coverage samples.

    EXT_framebuffer_multisample introduced the function
    RenderbufferStorageMultisampleEXT as a method of defining the
    storage parameters for a multisample render buffer. This function
    takes a <samples> parameter. Using rules provided by the
    specification, the <samples> parameter is resolved to an actual
    number of samples that is supported by the underlying hardware.
    EXT_framebuffer_multisample does not specify whether <samples>
    refers to coverage samples or color samples.

    This extension adds the function
    RenderbufferStorageMultisamplCoverageNV, which takes a
    <coverageSamples> parameter as well as a <colorSamples> parameter.

These two parameters give developers more fine grained control over
the quality of multisampled images.

New Procedures and Functions

```
void RenderbufferStorageMultisampleCoverageNV(
         enum target, sizei coverageSamples,
         sizei colorSamples, enum internalformat,
         sizei width, sizei height);
```

New Tokens

Accepted by the <pname> parameter of GetRenderbufferParameterivEXT:

```
RENDERBUFFER_COVERAGE_SAMPLES_NV         0x8CAB
RENDERBUFFER_COLOR_SAMPLES_NV            0x8E10
```

Additions to Chapter 2 of the OpenGL 2.0 Specification (OpenGL Operation)

None.

Additions to Chapter 3 of the OpenGL 2.0 Specification (Rasterization)

None.

Additions to Chapter 4 of the OpenGL 2.0 Specification (Per-Fragment Operations and the Frame Buffer)

Modification to 4.4.2.1 (Renderbuffer Objects)

Add, just above the definition of RenderbufferStorageMultisampleEXT:

"The command

```
void RenderbufferStorageMultisampleCoverageNV(
    enum target, sizei coverageSamples,
    sizei colorSamples, enum internalformat,
    sizei width, sizei height);
```

establishes the data storage, format, dimensions, number of coverage
samples, and number of color samples of a renderbuffer object's
image. <target> must be RENDERBUFFER_EXT. <internalformat> must be
RGB, RGBA, DEPTH_COMPONENT, STENCIL_INDEX, or one of the internal
formats from table 3.16 or table 2.nnn that has a base internal
format of RGB, RGBA, DEPTH_COMPONENT, or STENCIL_INDEX. <width>
and <height> are the dimensions in pixels of the renderbuffer. If
either <width> or <height> is greater than
MAX_RENDERBUFFER_SIZE_EXT, the error INVALID_VALUE is generated. If
the GL is unable to create a data store of the requested size, the
error OUT_OF_MEMORY is generated.

Upon success, RenderbufferStorageMultisampleCoverageNV deletes any
existing data store for the renderbuffer image and the contents of
the data store after calling
RenderbufferStorageMultisampleCoverageNV are undefined.
RENDERBUFFER_WIDTH_EXT is set to <width>, RENDERBUFFER_HEIGHT_EXT

is set to <height>, and RENDERBUFFER_INTERNAL_FORMAT_EXT is set to <internalformat>.

If <coverageSamples> is zero, then RENDERBUFFER_COVERAGE_SAMPLES_NV is set to zero. Otherwise <coverageSamples> represents a request for a desired minimum number of coverage samples. Since different implementations may support different coverage sample counts for multisampled rendering, the actual number of coverage samples allocated for the renderbuffer image is implementation dependent. However, the resulting value for RENDERBUFFER_COVERAGE_SAMPLES_NV is guaranteed to be greater than or equal to <coverageSamples> and no more than the next larger coverage sample count supported by the implementation.

If <colorSamples> is zero then RENDERBUFFER_COLOR_SAMPLES_NV is set to zero. Otherwise, <colorSamples> represents a request for a desired minimum number of colors samples. Since different implementations may support different color sample counts for multisampled rendering, the actual number of color samples allocated for the renderbuffer image is implementation dependent. Furthermore, a given implementation may support different color sample counts for each supported coverage sample count. The resulting value for RENDERBUFFER_COLOR_SAMPLES_NV is determined after resolving the value for RENDERBUFFER_COVERAGE_SAMPLES_NV. If the requested color sample count exceeds the maximum number of color samples supported by the implementation given the value of RENDERBUFFER_COVERAGE_SAMPLES_NV, the implementation will set RENDERBUFFER_COLOR_SAMPLES_NV to the highest supported value. Otherwise, the resulting value for RENDERBUFFER_COLOR_SAMPLES_NV is guaranteed to be greater than or equal to <colorSamples> and no more than the next larger color sample count supported by the implementation given the value of RENDERBUFFER_COVERAGE_SAMPLES_NV.

If <colorSamples> is greater than <coverageSamples>, the error INVALID_VALUE is generated.

If <coverageSamples> or <colorSamples> is greater than MAX_SAMPLES_EXT, the error INVALID_VALUE is generated.

If <coverageSamples> is greater than zero, and <colorSamples> is zero, RENDERBUFFER_COLOR_SAMPLES_NV is set to an implementation dependent value based on RENDERBUFFER_COVERAGE_SAMPLES_NV.

Modify the definition of RenderbufferStorageMultisampleEXT as follows:

"The command

```
    void RenderbufferStorageMultisampleEXT(
        enum target, sizei samples,
        enum internalformat,
        sizei width, sizei height);
```

is equivalent to calling

```
    RenderbufferStorageMultisamplesCoverageNv(target, samples, 0,
        internalforamt, width, height).
```

Modification to 4.4.4.2 (Framebuffer Completeness)

Modify the RENDERBUFFER_SAMPLES_EXT entry in the bullet list:

```
* The value of RENDERBUFFER_COVERAGE_SAMPLES_NV is the same for all
  attached images.
  { FRAMEBUFFER_INCOMPLETE_MULTISAMPLE }
```

Add an entry to the bullet list:

```
* The value of RENDERBUFFER_COLOR_SAMPLES_NV is the same for all
  attached images.
  { FRAMEBUFFER_INCOMPLETE_MULTISAMPLE_EXT }
```

Additions to Chapter 5 of the OpenGL 2.0 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 2.0 Specification (State and State Requests)

None.

Additions to Appendix A of the OpenGL 2.0 Specification (Invariance)

None.

Errors

The error INVALID_OPERATION is generated if RenderbufferStorageMultisampleCoverageNV is called and <colorSamples> is greater than <coverageSamples>

The error INVALID_VALUE is generated if RenderbufferStorageMultisampleCoverageNV is called and <coverageSamples> is greater than MAX_SAMPLES_EXT.

The error INVALID_VALUE is generated if RenderbufferStorageMultisampleCoverageNV is called and <colorSamples> is greater than MAX_SAMPLES_EXT.

New State

(add to table 8.nnn, "Renderbuffers (state per renderbuffer object)")

Get Value	Type	Get Command	Initial Value	Description	Section	Attribute
RENDERBUFFER_COVERAGE_SAMPLES_NV	Z+	GetRenderbufferParameterivEXT	0	Number of coverage samples used by the renderbuffer	4.4.2.1	-
RENDERBUFFER_COLOR_SAMPLES_NV	Z+	GetRenderbufferParameterivEXT	0	Number of color samples used by the renderbuffer	4.4.2.1	-

(modify RENDERBUFFER_SAMPLES_EXT entry in table 8.nnn)

Get Value	Type	Get Command	Initial Value	Description	Section	Attribute
RENDERBUFFER_SAMPLES_EXT	Z+	GetRenderbufferParameterivEXT	0	Alias for RENDERBUFFER_-COVERAGE_SAMPLES_NV	4.4.2.1	-

New Implementation Dependent State

None

Issues

(1) How should RenderbufferStorageMultisampleEXT be layered on top of RenderbufferStorageMultisampleCoverageNV?

RESOLVED. NVIDIA will expose this extension at the same time that EXT_framebuffer_multisample is exposed, so there will not be any issues with backward compatibility. However, some developers choose not to use vendor specific extensions. These developers should be able to make use of current and future hardware that differentiates between color and coverage samples. Since color samples are a subset of coverage samples, the <samples> parameter to RenderbufferStorageMultisampleEXT should be treated as a request for coverage samples. The implementation is free to choose the number of color samples used by the renderbuffer.

(2) <coverageSamples> is rounded up to the next highest number of samples supported by the implementation. How should <colorSamples> be rounded given that an implementation may not support all combinations of <coverageSamples> and <colorSamples>?

RESOLVED: It is a requirement that <coverageSamples> be compatible with the <samples> parameter to RenderbufferStorageMultisampleEXT. While it is desirable for <colorSamples> to resolve the same way as <coverageSamples>, this may not always be possible. An implementation may support a different maximum number of color samples for each coverage sample count. It would be confusing to set an error when <colorSamples> exceeds the maximum supported number of color samples for a given coverage sample count, because there is no mechanism to query or predict this behavior. Therefore, the implementation should round <colorSamples> down when it exceeds the maximum number of color samples supported with the given coverage sample count. Otherwise, <colorSamples> is rounded up to the next highest number of color samples supported by the implementation.

(3) Should a new query function be added so that an application can determine the maximum number of color samples supported with a given value of <coverageSamples>?

UNRESOLVED. Such a query would have to evaluate <coverageSamples>, and resolve it to an implementation supported value. The query would then return the maximum number of color samples supported given the resolved value of <coverageSamples>. There is no precedent for supporting a query of an implementation dependent value that requires complex evaluation of a parameter to the query. Adding such a query is unlikely.

An alternative query mechanism might involve a pair of queries. One query returns the maximum number of unique combinations of coverage samples and color samples supported by the implementation. A second query is used to enumerate these combinations. In the event that no such query mechanism is added, an application can still determin the set of unique and valid combinations of coverage samples and color samples.

An application wishing to implement such a query can do so by creating a set of multisample renderbuffers and querying their properties. A renderbuffer can be created for each (<coverageSamples>, <colorSamples>) pair where <coverageSamples> is in [1, MAX_SAMPLES_EXT], and <colorSamples> is in [1, <coverageSamples>]. The application can query RENDERBUFFER_COVERAGE_SAMPLES_NV and RENDERBUFFER_COLOR_SAMPLES_NV for each renderbuffer, using the results to identify the set of unique (<coverageSamples>, <colorSamples>) pairs supported by the implementation.

Revision History

None

Name

    NV_geometry_program4

Name Strings

    (none)

Contact

    Pat Brown, NVIDIA Corporation (pbrown 'at' nvidia.com)

Status

    Shipping for GeForce 8 Series (November 2006)

Version

```
    Last Modified Date:         11/06/2006
    NVIDIA Revision:            6
```

Number

    323

Dependencies

    OpenGL 1.1 is required.

    This extension is written against the OpenGL 2.0 specification.

    NV_gpu_program4 is required. This extension is supported if
    "GL_NV_gpu_program4" is found in the extension string.

    EXT_framebuffer_object interacts with this extension.

    EXT_framebuffer_blit interacts with this extension.

    EXT_texture_array interacts with this extension.

    ARB_texture_rectangle trivially affects the definition of this extension.

    EXT_texture_buffer_object trivially affects the definition of this
    extension.

    NV_primitive_restart trivially affects the definition of this extension.

Overview

    NV_geometry_program4 defines a new type of program available to be run on
    the GPU, called a geometry program. Geometry programs are run on full
    primitives after vertices are transformed, but prior to flat shading and
    clipping.

    A geometry program begins with a single primitive - a point, line, or
    triangle. Quads and polygons are allowed, but are decomposed into
    individual triangles prior to geometry program execution. It can read the

attributes of any of the vertex in the primitive and use them to generate new primitives. A geometry program has a fixed output primitive type, either a point, a line strip, or a triangle strip. It emits vertices (using the EMIT opcode) to define the output primitive. The attributes of emitted vertices are specified by writing to the same set of result bindings (e.g., "result.position") provided for vertex programs. Additionally, a geometry program can emit multiple disconnected primitives by using the ENDPRIM opcode, which is roughly equivalent to calling End and then Begin again. The primitives emitted by the geometry program are then clipped and then processed like an equivalent OpenGL primitive specified by the application.

This extension provides four additional primitive types: lines with adjacency, line strips with adjacency, separate triangles with adjacency, and triangle strips with adjacency. Some of the vertices specified in these new primitive types are not part of the ordinary primitives. Instead, they represent neighboring vertices that are adjacent to the two line segment end points (lines/strips) or the three triangle edges (triangles/tstrips). These "adjacency" vertices can be accessed by geometry programs and used to match up the outputs of the geometry program with those of neighboring primitives.

Additionally, geometry programs allow for layered rendering, where entire three-dimensional, cube map, or array textures (EXT_texture_array) can be bound to the current framebuffer. Geometry programs can use the "result.layer" binding to select a layer or cube map face to render to. Each primitive emitted by such a geometry program is rendered to the layer taken from its provoking vertex.

Since geometry programs expect a specific input primitive type, an error will occur if the application presents primtives of a different type. For example, if an enabled geometry program expects points, an error will occur at Begin() time, if a primitive mode of TRIANGLES is specified.

New Procedures and Functions

```
void ProgramVertexLimitNV(enum target, int limit);

void FramebufferTextureEXT(enum target, enum attachment,
                           uint texture, int level);
void FramebufferTextureLayerEXT(enum target, enum attachment,
                                uint texture, int level, int layer);
```

New Tokens

Accepted by the <cap> parameter of Disable, Enable, and IsEnabled, and by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

```
    GEOMETRY_PROGRAM_NV                             0x8C26
Accepted by the <pname> parameter of GetProgramivARB:

    MAX_PROGRAM_OUTPUT_VERTICES_NV                  0x8C27
    MAX_PROGRAM_TOTAL_OUTPUT_COMPONENTS_NV          0x8C28
    GEOMETRY_VERTICES_OUT_EXT                       0x8DDA
    GEOMETRY_INPUT_TYPE_EXT                         0x8DDB
    GEOMETRY_OUTPUT_TYPE_EXT                        0x8DDC

Accepted by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv,
and GetDoublev:

    MAX_GEOMETRY_TEXTURE_IMAGE_UNITS_EXT            0x8C29

Accepted by the <mode> parameter of Begin, DrawArrays, MultiDrawArrays,
DrawElements, MultiDrawElements, and DrawRangeElements:

    LINES_ADJACENCY_EXT                             0xA
    LINE_STRIP_ADJACENCY_EXT                        0xB
    TRIANGLES_ADJACENCY_EXT                         0xC
    TRIANGLE_STRIP_ADJACENCY_EXT                    0xD

Returned by CheckFramebufferStatusEXT:

    FRAMEBUFFER_INCOMPLETE_LAYER_TARGETS_EXT        0x8DA8
    FRAMEBUFFER_INCOMPLETE_LAYER_COUNT_EXT          0x8DA9

Accepted by the <pname> parameter of
GetFramebufferAttachmentParameterivEXT:

    FRAMEBUFFER_ATTACHMENT_LAYERED_EXT              0x8DA7
    FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER_EXT        0x8CD4

Accepted by the <cap> parameter of Enable, Disable, and IsEnabled, and by
the <pname> parameter of GetIntegerv, GetFloatv, GetDoublev, and
GetBooleanv:

    PROGRAM_POINT_SIZE_EXT                          0x8642

(Note:  The "EXT" tokens above are shared with the EXT_geometry_shader4
extension.)

(Note:  FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER is simply an alias for the
FRAMEBUFFER_ATTACHMENT_TEXTURE_3D_ZOFFSET_EXT token provided in
EXT_framebuffer_object.  This extension generalizes the notion of
"<zoffset>" to include layers of an array texture.)

(Note:  PROGRAM_POINT_SIZE_EXT is simply an alias for the
VERTEX_PROGRAM_POINT_SIZE token provided in OpenGL 2.0, which is itself an
alias for VERTEX_PROGRAM_POINT_SIZE_ARB provided by ARB_vertex_program.
Program-computed point sizes can be enabled if geometry programs are
enabled, even if no vertex program is used.)
```

Additions to Chapter 2 of the OpenGL 1.5 Specification (OpenGL Operation)

Modify Section 2.6.1 (Begin and End Objects), p. 13

(Add to end of section, p. 18)

(add figure)

```
    1 - - - 2----->3 - - - 4     1 - - - 2--->3--->4--->5 - - - 6

    5 - - - 6----->7 - - - 8

           (a)                             (b)
```

Figure X.1 (a) Lines with adjacency, (b) Line strip with adjacency. The vertices connected with solid lines belong to the main primitives; the vertices connected by dashed lines are the adjacent vertices that may be used in a geometry program.

Lines with Adjacency

Lines with adjacency are independent line segments where each endpoint has a corresponding "adjacent" vertex that can be accessed by a geometry program (Section 2.15). If geometry programs are disabled, the "adjacent" vertices are ignored.

A line segment is drawn from the 4i + 2nd vertex to the 4i + 3rd vertex for each i = 0, 1, ... , n-1, where there are 4n+k vertices between the Begin and End. k is either 0, 1, 2, or 3; if k is not zero, the final k vertices are ignored. For line segment i, the 4i + 1st and 4i + 4th vertices are considered adjacent to the 4i + 2nd and 4i + 3rd vertices, respectively. See Figure X.1.

Lines with adjacency are generated by calling Begin with the argument value LINES_ADJACENCY_EXT.

Line Strips with Adjacency

Line strips with adjacency are similar to line strips, except that each line segment has a pair of adjacent vertices that can be accessed by geometry programs (Section 2.15). If geometry programs are disabled, the "adjacent" vertices are ignored.

A line segment is drawn from the i + 2nd vertex to the i + 3rd vertex for each i = 0, 1, ..., n-1, where there are n+3 vertices between the Begin and End. If there are fewer than four vertices between a Begin and End, all vertices are ignored. For line segment i, the i + 1st and i + 4th vertices are considered adjacent to the i + 2nd and i + 3rd vertices, respectively. See Figure X.1.

Line strips with adjacency are generated by calling Begin with the argument value LINE_STRIP_ADJACENCY_EXT.

```
(add figure)
               2 - - - 3 - - - 4     8 - - - 9 - - - 10
                       ^\                    ^\
                 \     | \     |       \     | \     |
                       |  \                  |  \
                   \   |   \   |         \   |   \   |
                       |    \                |    \
                     \ |     \ |           \ |     \ |
                       |      v              |      v
                       1<------5             7<------11

                         \     |               \     |

                           \   |                 \   |

                             \ |                   \ |

                               6                     12
```

Figure X.2 Triangles with adjacency. The vertices connected with solid lines belong to the main primitive; the vertices connected by dashed lines are the adjacent vertices that may be used in a geometry program.

Triangles with Adjacency

Triangles with adjacency are similar to separate triangles, except that each triangle edge has an adjacent vertex that can be accessed by geometry programs (Section 2.15). If geometry programs are disabled, the "adjacent" vertices are ignored.

The 6i + 1st, 6i + 3rd, and 6i + 5th vertices (in that order) determine a triangle for each i = 0, 1, ..., n-1, where there are 6n+k vertices between the Begin and End. k is either 0, 1, 2, 3, 4, or 5; if k is non-zero, the final k vertices are ignored. For triangle i, the i + 2nd, i + 4th, and i + 6th vertices are considered adjacent to edges from the i + 1st to the i + 3rd, from the i + 3rd to the i + 5th, and from the i + 5th to the i + 1st vertices, respectively. See Figure X.2.

Triangles with adjacency are generated by calling Begin with the argument value TRIANGLES_ADJACENCY_EXT.

```
   (add figure)
                                 6                     6

                                 | \                   | \

                                 |   \                 |   \

                                 |     \               |     \
     2 - - - 3- - - >6   2 - - - 3------>7     2 - - - 3------>7- - - 10
             ^\                  ^^      |             ^^      ^^      |
       \     | \     |     \     | \     | \     \     | \     | \
             |  \                |  \    |             |  \    |  \    |
         \   |   \   |       \   |   \   |   \     \   |   \   |   \
             |    \                 |    \  |             |    \  |    \  |
           \ |     \ |         \ |     \ |     \     \ |     \ |     \
             |      v            |      vv             |      vv      v|
             1<------5           1<------5 - - - 8     1<------5<------9

               \     |             \     |               \     | \     |

                 \   |               \   |                 \   |   \   |

                   \ |                 \ |                   \ |     \ |

                     4                   4                     4       8

                                  6       10

                                  | \     | \

                                  |   \   |   \

                                  |     \ |     \
                          2 - - - 3------>7------>11
                                  ^^      ^^      |
                            \     | \     | \     | \
                                  |  \    |  \    |
                              \   |   \   |   \   |   \
                                  |    \  |    \  |
                                \ |     \ |     \ |     \
                                  |      vv      vv
                                  1<------5<------9 - - - 12

                                    \     | \     |

                                      \   |   \   |

                                        \ |     \ |

                                          4       8
```

Figure X.3 Triangle strips with adjacency. The vertices connected with solid lines belong to the main primitives; the vertices connected by dashed lines are the adjacent vertices that may be used in a geometry program.

Triangle Strips with Adjacency

Triangle strips with adjacency are similar to triangle strips, except that each triangle edge has an adjacent vertex that can be accessed by geometry programs (Section 2.15). If geometry programs are disabled, the "adjacent" vertices are ignored.

In triangle strips with adjacency, n triangles are drawn using 2 * (n+2) + k vertices between the Begin and End. k is either 0 or 1; if k is 1, the final vertex is ignored. If fewer than 6 vertices are specified between the Begin and End, the entire primitive is ignored. Table X.1 describes the vertices and order used to draw each triangle, and which vertices are considered adjacent to each edge of the triangle. See Figure X.3.

(add table)

	primitive vertices			adjacent vertices		
primitive	1st	2nd	3rd	1/2	2/3	3/1
only (i==0, n==1)	1	3	5	2	6	4
first (i==0)	1	3	5	2	7	4
middle (i odd)	2i+3	2i+1	2i+5	2i-1	2i+4	2i+7
middle (i even)	2i+1	2i+3	2i+5	2i-1	2i+7	2i+4
last (i==n-1, i odd)	2i+3	2i+1	2i+5	2i-1	2i+4	2i+6
last (i==n-1, i even)	2i+1	2i+3	2i+5	2i-1	2i+6	2i+4

Table X.1: Triangles generated by triangle strips with adjacency. Each triangle is drawn using the vertices in the "1st", "2nd", and "3rd" columns under "primitive vertices", in that order. The vertices in the "1/2", "2/3", and "3/1" columns under "adjacent vertices" are considered adjacent to the edges from the first to the second, from the second to the third, and from the third to the first vertex of the triangle, respectively. The six rows correspond to the six cases: the first and only triangle (i=0, n=1), the first triangle of several (i=0, n>0), "odd" middle triangles (i=1,3,5...), "even" middle triangles (i=2,4,6,...), and special cases for the last triangle inside the Begin/End, when i is either even or odd. For the purposes of this table, the first vertex specified after Begin is numbered "1" and the first triangle is numbered "0".

Triangle strips with adjacency are generated by calling Begin with the argument value TRIANGLE_STRIP_ADJACENCY_EXT.

Modify Section 2.14.1, Lighting (p. 59)

(modify fourth paragraph, p. 63) Additionally, vertex and geometry shaders and programs can operate in two-sided color mode, which is enabled and disabled by calling Enable or Disable with the symbolic value VERTEX_PROGRAM_TWO_SIDE. When a vertex or geometry shader is active, the shaders can write front and back color values to the gl_FrontColor, gl_BackColor, gl_FrontSecondaryColor and gl_BackSecondaryColor outputs. When a vertex or geometry program is active, programs can write front and back colors using the available color result bindings. When a vertex or geometry shader or program is active and two-sided color mode is enabled, the GL chooses between front and back colors, as described below. If two-sided color mode is disabled, the front color output is always selected.

Insert New Section 2.14.6, Geometry Programs (between 2.14.5, Color Index Lighting and 2.14.6, Clamping and Masking, p. 69)

Section 2.14.6, Geometry Programs

Each primitive may be optionally transformed by a geometry program. Geometry programs are enabled by calling Enable with the value GEOMETRY_PROGRAM_NV. A geometry program takes a single input primitive and generates vertices to be arranged into one or more output primitives. The original input primitive is discarded, and the output primitives are processed in order by the remainder of the GL pipeline.

Section 2.14.6.1, Geometry Program Input Primitives

A geometry program can operate on one of five input primitive types, as specified by the mandatory "PRIMITIVE_IN" declaration. Depending on the input primitive type, one to six vertices are available when the program is executed. A geometry program will fail to load unless it contains exactly one such declaration.

Each input primitive type supports only a subset of the primitives provided by the GL. If geometry programs are enabled, Begin, or any function that implicitly calls Begin, will produce an INVALID_OPERATION error if the <mode> parameter is incompatible with the input primitive type of the current geometry program.

The supported input primitive types are:

Points (POINTS)

Geometry programs that operate on points are valid only for the POINTS primitive type. There is a only a single vertex available for each program invocation: "vertex[0]" refers to the single point.

Lines (LINES)

Geometry programs that operate on line segments are valid only for the LINES, LINE_STRIP, and LINE_LOOP primitive types. There are two vertices available for each program invocation: "vertex[0]" and "vertex[1]" refer to the beginning and end of the line segment.

Lines with Adjacency (LINES_ADJACENCY)

Geometry programs that operate on line segments with adjacent vertices are valid only for the LINES_ADJACENCY_EXT and LINE_STRIP_ADJACENCY_EXT primitive types. There are four vertices available for each program invocation. "vertex[1]" and "vertex[2]" refer to the beginning and end of the line segment. "vertex[0]" and "vertex[3]" refer to the vertices adjacent to the beginning and end of the line segment, respectively.

Triangles (TRIANGLES)

Geometry programs that operate on triangles are valid for the TRIANGLES, TRIANGLE_STRIP, TRIANGLE_FAN, QUADS, QUAD_STRIP, and POLYGON primitive types.

When used with a geometry program that operates on triangles, QUADS, QUAD_STRIP, and POLYGON primitives are decomposed into triangles in an unspecified, implementation-dependent manner. For convex polygons (already required in the core GL specification), this decomposition satisfies three properties:

* the collection of triangles fully covers the area of the original primitive,

* no two triangles in the decomposition overlap, and

* the orientation of each triangle is consistent with the orientation of the original primitive.

For such primitives, the program is executed once for each triangle in the decomposition.

There are three vertices available for each program invocation. "vertex[0]", "vertex[1]", and "vertex[2]", refer to the first, second, and third vertex of the triangle, respectively.

Triangles with Adjacency (TRIANGLES_ADJACENCY)

Geometry programs that operate on triangles with adjacent vertices are valid for the TRIANGLES_ADJACENCY_EXT and TRIANGLE_STRIP_ADJACENCY_EXT primitive types. There are six vertices available for each program invocation. "vertex[0]", "vertex[2]", and "vertex[4]" refer to the first, second, and third vertex of the triangle respectively. "vertex[1]", "vertex[3]", and "vertex[5]" refer to the vertices adjacent to the edges from the first to the second vertex, from the second to the third vertex, and from the third to the first vertex, respectively.

Section 2.14.6.2, Geometry Program Output Primitives

A geometry program can generate primitives of one of three types, as specified by the mandatory "PRIMITIVE_OUT" declaration. A geometry program will fail to load unless it contains exactly one such declaration.

The supported output primitive types are points (POINTS), line strips (LINE_STRIP), and triangle strips (TRIANGLE_STRIP). The vertices output by the geometry program are decomposed into points, lines, or triangles based on the output primitive type in the manner described in section 2.6.1.

Section 2.14.6.3, Geometry Program Execution Environment

Geometry programs execute using the instruction set documented in the GL_NV_gpu_program4 extension specification and in a manner similar to vertex programs. Each vertex attribute access must identify the vertex number being accessed. For example, "vertex[1].position" identifies the transformed position of "vertex[1]" as specified in teh description of the input primitive type. Output vertices are specified by writing to vertex result variables in the same manner as done by vertex programs.

The special instruction "EMIT" specifies that a vertex is completed. A vertex is added to the current output primitive using the current values of the vertex result variables. The values of any unwritten result variables (or components) are undefined.

After an EMIT instruction is completed, the current values of all vertex result variables become undefined. If a program wants to ensure that the same result is used for every vertex written by the program, it is necessary to write the corresponding value once per vertex.

The special instruction "ENDPRIM" specifies that the current output primitive should be completed and a new output primitive should be started. A geometry program starts with an output primitive containing no vertices. When a geometry program terminates, the current output primitive is automatically completed. ENDPRIM has no effect if the geometry program's output primitive type is POINTS.

When a primitive generated by a geometry program is completed, the vertices added by the EMIT instruction are decomposed into points, lines, or triangles according to the output primitive type in the manner described in Section 2.8.1. The resulting primitives are then clipped and rasterized. If the number of vertices emitted by the geometry program is not sufficient to produce a single primitive, nothing is drawn.

Like vertex and fragment programs, geometry programs can access textures. The maximum number of texture image units that can be accessed by a geometry program is given by the value of MAX_GEOMETRY_TEXTURE_IMAGE_UNITS_EXT.

Section 2.14.6.4, Geometry Program Output Limits

A geometry program may not emit an limited in the number of vertices per invocation. Each geometry program must declare a vertex limit, which is the maximum number of vertices that the program can ever produce. The vertex limit is specified using the "VERTICES_OUT" declaration. A geometry program will fail to load unless it contains exactly one such declaration.

There are two implementation-dependent limits that limit the total number of vertices that a program can emit. First, the vertex limit may not exceed the value of MAX_PROGRAM_OUTPUT_VERTICES_NV. Second, product of the vertex limit and the number of result variable components written by the program (PROGRAM_RESULT_COMPONENTS_NV, as described in section 2.X.3.5 of NV_gpu_program4) may not exceed the value of MAX_PROGRAM_TOTAL_OUTPUT_COMPONENTS_NV. A geometry program will fail to load if its maximum vertex count or maximum total component count exceeds the implementation-dependent limit. The limits may be queried by calling GetProgramiv with a <target> of GEOMETRY_PROGRAM_NV. Note that the maximum number of vertices that a geometry program can emit may be much lower than MAX_PROGRAM_OUTPUT_VERTICES_NV if the program writes a large number of result variable components.

After a geometry program is compiled, the vertex limit may be changed using the command

```
  void ProgramVertexLimitNV(enum target, int limit);
```

<target> must be GEOMETRY_PROGRAM_NV. <limit> is the new vertex limit, which must satisfy the two rules described above. The error INVALID_VALUE is generated if <limit> is less than or equal to zero, <limit> is greater than or equal to MAX_PROGRAM_OUTPUT_VERTICES_NV, or if the total number of components emitted would exceed MAX_PROGRAM_TOTAL_OUTPUT_COMPONENTS_NV. The error INVALID_OPERATION is generated if the current geometry program has not been successfully loaded.

When a program executes, the number of vertices it emits should not exceed the vertex limit. Once a geometry program emits a number of vertices equal to the vertex limit, subsequent EMIT instructions may or may not have any effect.

Modify Section 2.X.2, Program Grammar

(replace third paragraph)

Geometry programs are required to begin with the header string "!!NVgp4.0". This header string identifies the subsequent program body as being a geometry program and indicates that it should be parsed according to the base NV_gpu_program4 grammar plus the additions below. Program string parsing begins with the character immediately following the header string.

(add the following grammar rules to the NV_gpu_program4 base grammar)

```
<declSequence>          ::= <declaration> <declSequence>

<instruction>           ::= <SpecialInstruction>

<attribUseV>            ::= <attribVarName> <arrayMem> <arrayMem>
                            <swizzleSuffix>

<attribUseS>            ::= <attribVarName> <arrayMem> <arrayMem>
                            <scalarSuffix>

<attribUseVNS>          ::= <attribVarName> <arrayMem> <arrayMem>

<resultUseW>            ::= <resultVarName> <arrayMem> <optWriteMask>
                          | <resultColor> <optWriteMask>
                          | <resultColor> "." <colorType> <optWriteMask>
                          | <resultColor> "." <faceType> <optWriteMask>
                          | <resultColor> "." <faceType> "." <colorType>
                            "." <optWriteMask>

<resultUseD>            ::= <resultColor>
                          | <resultColor> "." <colorType>
                          | <resultMulti>

<declaration>           ::= "PRIMITIVE_IN" <declPrimInType>
                          | "PRIMITIVE_OUT" <declPrimOutType>
                          | "VERTICES_OUT" <int>
```

```
    <declPrimInType>        ::= "POINTS"
                              | "LINES"
                              | "LINES_ADJACENCY"
                              | "TRIANGLES"
                              | "TRIANGLES_ADJACENCY"

    <declPrimOutType>       ::= "POINTS"
                              | "LINE_STRIP"
                              | "TRIANGLE_STRIP"

    <SpecialInstruction>    ::= "EMIT"
                              | "ENDPRIM"

    <attribBasic>           ::= <vtxPrefix> "position"
                              | <vtxPrefix> "fogcoord"
                              | <vtxPrefix> "pointsize"
                              | <attribTexCoord> <optArrayMemAbs>
                              | <attribClip> <arrayMemAbs>
                              | <attribGeneric> <arrayMemAbs>
                              | "primitive" "." "id"

    <attribColor>           ::= <vtxPrefix> "color"

    <attribMulti>           ::= <attribTexCoord> <arrayRange>
                              | <attribClip> <arrayRange>
                              | <attribGeneric> <arrayRange>

    <attribTexCoord>        ::= <vtxPrefix> "texcoord"

    <attribClip>            ::= <vtxPrefix> "clip"

    <attribGeneric>         ::= <vtxPrefix> "attrib"

    <vtxPrefix>             ::= "vertex" <optArrayMemAbs>

    <resultBasic>           ::= <resPrefix> "position"
                              | <resPrefix> "fogcoord"
                              | <resPrefix> "pointsize"
                              | <resPrefix> "primid"
                              | <resPrefix> "layer"
                              | <resultTexCoord> <optArrayMemAbs>
                              | <resultClip> <arrayMemAbs>
                              | <resultGeneric> <arrayMemAbs>

    <resultColor>           ::= <resPrefix> "color"

    <resultMulti>           ::= <resultTexCoord> <arrayRange>
                              | <resultClip> <arrayRange>
                              | <resultGeneric> <arrayRange>

    <resultTexCoord>        ::= <resPrefix> "texcoord"

    <resultClip>            ::= <resPrefix> "clip"

    <resultGeneric>         ::= <resPrefix> "attrib"
```

```
<resPrefix>             ::= "result" "."
```

(add the following subsection to section 2.X.3.2, Program Attribute Variables)

Geometry program attribute variables describe the attributes of each transformed vertex accessible to the geometry program. Most attributes correspond to the per-vertex results generated by vertex program execution or fixed-function vertex processing. The "primitive.id" attribute is generated specially, as described below.

If vertex programs are enabled, attributes will be obtained from the per-vertex outputs of the vertex program used to generate the vertex in question. Geometry program attributes should be read using the same component data type used to write the corresponding vertex program results. The value of any attribute corresponding to a vertex output not written by the vertex program is undefined.

If vertex programs are disabled, attributes will be obtained from the values computed by fixed-function vertex processing. All attributes, except for the primitive ID should be read as floating-point values in this case.

Geometry Vertex Binding	Components	Description
vertex[m].position	(x,y,z,w)	clip coordinates
vertex[m].color	(r,g,b,a)	front primary color
vertex[m].color.primary	(r,g,b,a)	front primary color
vertex[m].color.secondary	(r,g,b,a)	front secondary color
vertex[m].color.front	(r,g,b,a)	front primary color
vertex[m].color.front.primary	(r,g,b,a)	front primary color
vertex[m].color.front.secondary	(r,g,b,a)	front secondary color
vertex[m].color.back	(r,g,b,a)	back primary color
vertex[m].color.back.primary	(r,g,b,a)	back primary color
vertex[m].color.back.secondary	(r,g,b,a)	back secondary color
vertex[m].fogcoord	(f,-,-,-)	fog coordinate
vertex[m].pointsize	(s,-,-,-)	point size
vertex[m].texcoord	(s,t,r,q)	texture coordinate, unit 0
vertex[m].texcoord[n]	(s,t,r,q)	texture coordinate, unit n
vertex[m].attrib[n]	(x,y,z,w)	generic interpolant n
vertex[m].clip[n]	(d,-,-,-)	clip plane distance
vertex[m].texcoord[n..o]	(s,t,r,q)	array of texture coordinates
vertex[m].attrib[n..o]	(x,y,z,w)	array of generic interpolants
vertex[m].clip[n..o]	(d,-,-,-)	array of clip distances
vertex[m].id	(id,-,-,-)	vertex id
primitive.id	(id,-,-,-)	primitive number

Table X.2, Geometry Program Attribute Bindings. <m> refers to a vertex number, while <n>, and <o> refer to integer constants. Only the "vertex[m].texcoord" and "vertex.attrib" bindings are available in arrays.

For bindings that include "vertex[m]", <m> identifies the vertex number whose attributes are used for the binding. For bindings in explicit variable declarations, "[m]" is optional. If "[m]" is specified, <m> must be an integer constant and must be in the valid range of vertices supported for the input primitive type. If "[m]" is not specified, the declared variable is accessed as an array, with the first array index specifying the vertex number. If such a variable is declared an array, it must have a second array index to identify the individual array element. For bindings used directly in instructions, "[m]" is required and must be an integer constant specifying a vertex number. The following examples illustrate various legal and illegal geometry program bindings and their meanings.

```
  ATTRIB pos = vertex.position;
  ATTRIB pos2 = vertex[2].position;
  ATTRIB texcoords[] = { vertex.texcoord[0..3] };
  ATTRIB tcoords1[4] = { vertex[1].texcoord[1..4] };
  INT TEMP A0;
  ...
  MOV R0, pos[1];                   # position of vertex 1
  MOV R0, vertex[1].position;       # position of vertex 1
  MOV R0, pos2;                     # position of vertex 2
  MOV R0, texcoords[A0.x][1];       # texcoord 1 of vertex A0.x
  MOV R0, texcoords[A0.x][A0.y];    # texcoord A0.y of vertex A0.x
  MOV R0, tcoords1[2];              # texcoord 3 of vertex 1
  MOV R0, vertex[A0.x].texcoord[1]; # ILLEGAL allowed -- vertex number
                                    #    must be constant here.
```

If a geometry attribute binding matches "vertex[m].position", the "x", "y", "z" and "w" components of the geometry attribute variable are filled with the "x", "y", "z", and "w" components, respectively, of the transformed position of vertex <m>, in clip coordinates.

If a geometry attribute binding matches any binding in Table X.2 beginning with "vertex[m].color", the "x", "y", "z", and "w" components of the geometry attribute variable are filled with the "r", "g", "b", and "a" components, respectively, of the corresponding color of vertex <m>. Bindings containing "front" and "back" refer to the front and back colors, respectively. Bindings containing "primary" and "secondary" refer to primary and secondary colors, respectively. If face or color type is omitted in the binding, the binding is treated as though "front" and "primary", respectively, were specified.

If a geometry attribute binding matches "vertex[m].fogcoord", the "x" component of the geometry attribute variable is filled with the fog coordinate of vertex <m>. The "y", "z", and "w" components are undefined.

If a geometry attribute binding matches "vertex[m].pointsize", the "x" component of the geometry attribute variable is filled with the point size of vertex <m> computed by the vertex program. For fixed-function vertex processing, the point size attribute is undefined. The "y", "z", and "w" components are always undefined.

If a geometry attribute binding matches "vertex[m].texcoord" or "vertex[m].texcoord[n]", the "x", "y", "z", and "w" coordinates of the geometry attribute variable are filled with the "s", "t", "r", and "q" coordinates of texture coordinate set <n> of vertex <m>. If <n> is omitted, texture coordinate set zero is used.

If a geometry attribute binding matches "vertex[m].attrib[n]", the "x", "y", "z", and "w" components of the geometry attribute variable are filled with the "x", "y", "z", and "w" coordinates of generic interpolant <n> of vertex <m>. All generic interpolants will be undefined when used with fixed-function vertex processing.

If a geometry attribute binding matches "vertex[m].clip[n]", the "x" component of the geometry attribute variable is filled the clip distance of vertex <m> for clip plane <n>, as written by the vertex program. If fixed-function vertex processing or position-invariant vertex programs are used, the clip distance is obtained by computing the per-clip plane dot product:

```
  (p_1' p_2' p_3' p_4') dot (x_e y_e z_e w_e),
```

at the vertex location, as described in section 2.12. The clip distance for clip plane <n> is undefined if clip plane <n> is disabled. The "y", "z", and "w" components of the attribute are undefined.

If a geometry attribute binding matches "vertex[m].texcoord[n..o]", "vertex[m].attrib[n..o]", or "vertex[m].clip[n..o]", a sequence of 1+<o>-<n> texture coordinate bindings is created. For texture coordinate bindings, it is as though the sequence "vertex[m].texcoord[n], vertex[m].texcoord[n+1], ... vertex[m].texcoord[o]" were specfied. These bindings are available only in explicit declarations of array variables. A program will fail to load if <n> is greater than <o>.

If a geometry attribute binding matches "vertex[m].id", the "x" component is filled with the vertex ID. If a vertex program is currently active, the attribute variable is filled with the vertex ID result written by the vertex program. If fixed-function vertex processing is used, the vertex ID is undefined. The "y", "z", and "w" components of the attribute are undefined.

If a geometry attribute binding matches "primitive.id", the "x" component is filled with the number of primitives received by the GL since the last time Begin was called (directly or indirectly via vertex array functions). The first primitive generated after a Begin is numbered zero, and the primitive ID counter is incremented after every individual point, line, or polygon primitive is processed. For QUADS and QUAD_STRIP primitives that are decomposed into triangles, the primitive ID is incremented after each complete quad is processed. For POLYGON primitives, the primitive ID counter is zero. Restarting a primitive topology using the primitive restart index has no effect on the primitive ID counter. The "y", "z", and "w" components of the variable are always undefined.

(add the following subsection to section 2.X.3.5, Program Results.)

Geometry programs emit vertices, and the set of result variables available to such programs correspond to the attributes of each emitted vertex. The set of allowable result variable bindings for geometry programs is given in Table X.3.

Binding	Components	Description
result.position	(x,y,z,w)	position in clip coordinates
result.color	(r,g,b,a)	front-facing primary color
result.color.primary	(r,g,b,a)	front-facing primary color
result.color.secondary	(r,g,b,a)	front-facing secondary color
result.color.front	(r,g,b,a)	front-facing primary color
result.color.front.primary	(r,g,b,a)	front-facing primary color
result.color.front.secondary	(r,g,b,a)	front-facing secondary color
result.color.back	(r,g,b,a)	back-facing primary color
result.color.back.primary	(r,g,b,a)	back-facing primary color
result.color.back.secondary	(r,g,b,a)	back-facing secondary color
result.fogcoord	(f,*,*,*)	fog coordinate
result.pointsize	(s,*,*,*)	point size
result.texcoord	(s,t,r,q)	texture coordinate, unit 0
result.texcoord[n]	(s,t,r,q)	texture coordinate, unit n
result.attrib[n]	(x,y,z,w)	generic interpolant n
result.clip[n]	(d,*,*,*)	clip plane distance
result.texcoord[n..o]	(s,t,r,q)	texture coordinates n thru o
result.attrib[n..o]	(x,y,z,w)	generic interpolants n thru o
result.clip[n..o]	(d,*,*,*)	clip distances n thru o
result.primid	(id,*,*,*)	primitive id
result.layer	(l,*,*,*)	layer for cube/array/3D FBOs

Table X.3: Geometry Program Result Variable Bindings.
Components labeled "*" are unused.

If a result variable binding matches "result.position", updates to the "x", "y", "z", and "w" components of the result variable modify the "x", "y", "z", and "w" components, respectively, of the transformed vertex's clip coordinates. Final window coordinates will be generated for the vertex as described in section 2.14.4.4.

If a result variable binding match begins with "result.color", updates to the "x", "y", "z", and "w" components of the result variable modify the "r", "g", "b", and "a" components, respectively, of the corresponding vertex color attribute in Table X.3. Color bindings that do not specify "front" or "back" are consided to refer to front-facing colors. Color bindings that do not specify "primary" or "secondary" are considered to refer to primary colors.

If a result variable binding matches "result.fogcoord", updates to the "x" component of the result variable set the transformed vertex's fog coordinate. Updates to the "y", "z", and "w" components of the result variable have no effect.

If a result variable binding matches "result.pointsize", updates to the "x" component of the result variable set the transformed vertex's point size. Updates to the "y", "z", and "w" components of the result variable have no effect.

If a result variable binding matches "result.texcoord" or "result.texcoord[n]", updates to the "x", "y", "z", and "w" components of the result variable set the "s", "t", "r" and "q" components, respectively, of the transformed vertex's texture coordinates for texture unit <n>. If "[n]" is omitted, texture unit zero is selected.

If a result variable binding matches "result.attrib[n]", updates to the "x", "y", "z", and "w" components of the result variable set the "x", "y", "z", and "w" components of the generic interpolant <n>.

If a result variable binding matches "result.clip[n]", updates to the "x" component of the result variable set the clip distance for clip plane <n>.

If a result variable binding matches "result.texcoord[n..o]", "result.attrib[n..o]", or "result.clip[n..o]", a sequence of 1+<o>-<n> bindings is created. For texture coordinates, it is as though the sequence "result.texcoord[n], result.texcoord[n+1], ... result.texcoord[o]" were specfied. These bindings are available only in explicit declarations of array variables. A program will fail to load if <n> is greater than <o>.

If a result variable binding matches "result.primid", updates to the "x" component of the result variable provide a single integer that serves as a primitive identifier. The written primitive ID is available to fragment programs using the "primitive.id" attribute binding. If a fragment program using primitive IDs is active and a geometry program is also active, the geometry program must write "result.primid" or the primitive ID number is undefined.

If a result variable binding matches "result.layer", updates to the "x" component of the result variable provide a single integer that serves as a layer selector for layered rendering (section 2.14.6.5). The layer must be written as an integer value; writing a floating-point layer number will produce undefined results.

(modify Table X.13 in section 2.X.4, Program Instructions, to include the following.)

```
            Modifiers
Instruction F I C S H D  Inputs     Out  Description
----------- - - - - - -  ---------- ---  --------------------------------
EMIT        - - - - - -  -          -    emit vertex
ENDPRIM     - - - - - -  -          -    end of primitive
```

(add the following subsection to section 2.X.5, Program Options.)

Section 2.X.5.Y, Geometry Program Options

No options are supported at present for geometry programs.

(add the following subsection to section 2.X.6, Program Declarations.)

Section 2.X.6.Y, Geometry Program Declarations

Geometry programs support three types of declaration statements, as described below. Each of the three must be included exactly once in the geometry program.

- Input Primitive Type (PRIMITIVE_IN)

The PRIMITIVE_IN statement declares the type of primitives seen by a geometry program. The single argument must be one of "POINTS", "LINES", "LINES_ADJACENCY", "TRIANGLES", or "TRIANGLES_ADJACENCY".

- Output Primitive Type (PRIMITIVE_OUT)

The PRIMITIVE_OUT statement declares the type of primitive emitted by a geometry program. The single argument must be one of "POINTS", "LINE_STRIP", or "TRIANGLE_STRIP".

- Maximum Vertex Count (VERTICES_OUT)

The VERTICES_OUT statement declares the maximum number of vertices that may be emitted by a geometry program. The single argument must be a positive integer. A vertex program that emits more than the specified number of vertices may terminate abnormally.

(add the following subsections to section 2.X.7, Program Instruction Set.)

Section 2.X.7.Z, EMIT: Emit Vertex

The EMIT instruction emits a new vertex to be added to the current output primitive of a geometry program. The attributes of the emitted vertex are given by the current values of the vertex result variables. After the EMIT instruction completes, a new vertex is started and all result variables become undefined.

Section 2.X.7.Z, ENDPRIM: End of Primitive

A geometry program can emit multiple primitives in a single invocation. The ENDPRIM instruction is used in a geometry program to signify the end of the current primitive and the beginning of a new primitive of the same type. The effect of ENDPRIM is roughly equivalent to calling End followed by a new Begin, where the primitive mode is specified in the text of the geometry program.

Like End, the ENDPRIM instruction does not emit a vertex. Any result registers written prior to an ENDPRIM instruction are unchanged, and will be used in the vertex specified by the next EMIT instruction if they are not overwritten first.

When geometry program execution completes, the current primitive is automatically terminated. It is not necessary to include an ENDPRIM instruction if the geometry program writes only a single primitive.

Additions to Chapter 3 of the OpenGL 1.5 Specification (Rasterization)

Modify Section 3.3, Points (p. 95)

(replace all Section 3.3 text on p. 95) A point is drawn by generating a set of fragments in the shape of a square or circle centered around the vertex of the point. Each vertex has an associated point size that controls the size of that square or circle.

If no vertex or geometry program is active, the size of the point is controlled by

```
  void PointSize(float size);
```

<size> specifies the requested size of a point. The default value is 1.0. A value less than or equal to zero results in the error INVALID_VALUE.

The requested point size is multiplied with a distance attenuation factor, clamped to a specified point size range, and further clamped to the implementation-dependent point size range to produce the derived point size:

```
   derived size = clamp(size * sqrt(1/(a+b*d+c*d^2)))
```

where d is the eye-coordinate distance from the eye, (0,0,0,1) in eye coordinates, to the vertex, and a, b, and c are distance attenuation function coefficients.

If a vertex or geometry program is active, the derived size depends on the per-vertex point size mode enable. Per-vertex point size mode is enabled or disabled by calling Enable or Disable with the symbolic value PROGRAM_POINT_SIZE_EXT. If per-vertex point size is enabled and a geometry program is active, the point size is taken from the point size emitted by the geometry program. If per-vertex point size is enabled an no geometry program is active, the point size is taken from the point size result of the vertex program. Otherwise, the point size is taken from the <size> value provided to PointSize, with no distance attenuation applied. In all cases, the point size is clamped to the implementation-dependent point size range.

If multisampling is not enabled, the derived size is passed on to rasterization as the point width. ...

Additions to Chapter 4 of the OpenGL 1.5 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 1.5 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 1.5 Specification (State and State Requests)

None.

Additions to Appendix A of the OpenGL 1.5 Specification (Invariance)

None.

Additions to the AGL/GLX/WGL Specifications

None.

GLX Protocol

None.

Errors

The error INVALID_OPERATION is generated if Begin, or any command that implicitly calls Begin, is called when geometry program mode is enabled and the currently bound geometry program object does not contain a valid geometry program.

The error INVALID_OPERATION is generated if Begin, or any command that implicitly calls Begin, is called when geometry program mode is enabled and:

* the input primitive type of the current geometry program is POINTS and <mode> is not POINTS,

* the input primitive type of the current geometry program is LINES and <mode> is not LINES, LINE_STRIP, or LINE_LOOP,

* the input primitive type of the current geometry program is TRIANGLES and <mode> is not TRIANGLES, TRIANGLE_STRIP, TRIANGLE_FAN, QUADS, QUAD_STRIP, or POLYGON,

* the input primitive type of the current geometry program is LINES_ADJACENCY and <mode> is not LINES_ADJACENCY_EXT or LINE_STRIP_ADJACENCY_EXT, or

* the input primitive type of the current geometry program is TRIANGLES_ADJACENCY and <mode> is not TRIANGLES_ADJACENCY_EXT or TRIANGLE_STRIP_ADJACENCY_EXT.

The error INVALID_ENUM is generated if GetProgramivARB is called with a <pname> of MAX_PROGRAM_OUTPUT_VERTICES_NV or MAX_PROGRAM_TOTAL_OUTPUT_COMPONENTS_NV and the target isn't GEOMETRY_PROGRAM_NV.

Dependencies on EXT_framebuffer_object

If EXT_framebuffer_object (or similar functionality) is not supported, the "result.layer" binding should be removed. "FramebufferTextureEXT" and "FramebufferTextureLayerEXT" should be removed from "New Procedures and Functions", and FRAMEBUFFER_ATTACHMENT_LAYERED_EXT, FRAMEBUFFER_INCOMPLETE_LAYER_TARGETS_EXT, and FRAMEBUFFER_INCOMPLETE_LAYER_COUNT_EXT should be removed from "New Tokens".

Otherwise, this extension modifies EXT_framebuffer_object to add the notion of layered framebuffer attachments and framebuffers that can be used in conjunction with geometry programs to allow programs to direct primitives to a face of a cube map or layer of a three-dimensional texture or one- or two-dimensional array texture. The layer used for rendering can be selected by the geometry program at run time.

(insert before the end of Section 4.4.2, Attaching Images to Framebuffer Objects)

There are several types of framebuffer-attachable images:

* the image of a renderbuffer object, which is always two-dimensional,

* a single level of a one-dimensional texture, which is treated as a two-dimensional image with a height of one,

* a single level of a two-dimensional or rectangle texture,

* a single face of a cube map texture level, which is treated as a two-dimensional image, or

* a single layer of a one- or two-dimensional array texture or three-dimensional texture, which is treated as a two-dimensional image.

Additionally, an entire level of a three-dimensional texture, cube map texture, or one- or two-dimensional array texture can be attached to an attachment point. Such attachments are treated as an array of two-dimensional images, arranged in layers, and the corresponding attachment point is considered to be layered.

(replace section 4.4.2.3, "Attaching Texture Images to a Framebuffer")

GL supports copying the rendered contents of the framebuffer into the images of a texture object through the use of the routines CopyTexImage{1D|2D}, and CopyTexSubImage{1D|2D|3D}. Additionally, GL supports rendering directly into the images of a texture object.

To render directly into a texture image, a specified level of a texture object can be attached as one of the logical buffers of the currently bound framebuffer object by calling:

```
  void FramebufferTextureEXT(enum target, enum attachment,
                             uint texture, int level);
```

<target> must be FRAMEBUFFER_EXT. <attachment> must be one of the attachment points of the framebuffer listed in table 1.nnn.

If <texture> is zero, any image or array of images attached to the attachment point named by <attachment> is detached, and the state of the attachment point is reset to its initial values. <level> is ignored if <texture> is zero.

If <texture> is non-zero, FramebufferTextureEXT attaches level <level> of the texture object named <texture> to the framebuffer attachment point named by <attachment>. The error INVALID_VALUE is generated if <texture> is not the name of a texture object, or if <level> is not a supported texture level number for textures of the type corresponding to <target>. The error INVALID_OPERATION is generated if <texture> is the name of a buffer texture.

If <texture> is the name of a three-dimensional texture, cube map texture, or one- or two-dimensional array texture, the texture level attached to the framebuffer attachment point is an array of images, and the framebuffer attachment is considered layered.

The command

```
  void FramebufferTextureLayerEXT(enum target, enum attachment,
                                  uint texture, int level, int layer);
```

operates like FramebufferTextureEXT, except that only a single layer of the texture level, numbered <layer>, is attached to the attachment point. If <texture> is non-zero, the error INVALID_VALUE is generated if <layer> is negative, or if <texture> is not the name of a texture object. The error INVALID_OPERATION is generated unless <texture> is zero or the name of a three-dimensional or one- or two-dimensional array texture.

The command

```
  void FramebufferTextureFaceEXT(enum target, enum attachment,
                                 uint texture, int level, enum face);
```

operates like FramebufferTextureEXT, except that only a single face of a cube map texture, given by <face>, is attached to the attachment point. <face> is one of TEXTURE_CUBE_MAP_POSITIVE_X, TEXTURE_CUBE_MAP_NEGATIVE_X, TEXTURE_CUBE_MAP_POSITIVE_Y, TEXTURE_CUBE_MAP_NEGATIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z, TEXTURE_CUBE_MAP_NEGATIVE_Z. If <texture> is non-zero, the error INVALID_VALUE is generated if <texture> is not the name of a texture object. The error INVALID_OPERATION is generated unless <texture> is zero or the name of a cube map texture.

The command

```
  void FramebufferTexture1DEXT(enum target, enum attachment,
                               enum textarget, uint texture, int level);
```

operates identically to FramebufferTextureEXT, except for two additional restrictions. If <texture> is non-zero, the error INVALID_ENUM is generated if <textarget> is not TEXTURE_1D and the error INVALID_OPERATION is generated unless <texture> is the name of a one-dimensional texture.

The command

```
  void FramebufferTexture2DEXT(enum target, enum attachment,
                               enum textarget, uint texture, int level);
```

operates similarly to FramebufferTextureEXT. If <textarget> is TEXTURE_2D or TEXTURE_RECTANGLE_ARB, <texture> must be zero or the name of a two-dimensional or rectangle texture. If <textarget> is TEXTURE_CUBE_MAP_POSITIVE_X, TEXTURE_CUBE_MAP_NEGATIVE_X, TEXTURE_CUBE_MAP_POSITIVE_Y, TEXTURE_CUBE_MAP_NEGATIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z, or TEXTURE_CUBE_MAP_NEGATIVE_Z, <texture> must be zero or the name of a cube map texture. For cube map textures, only the single face of the cube map texture level given by <textarget> is attached. The error INVALID_ENUM is generated if <texture> is not zero and <textarget> is not one of the values enumerated above. The error INVALID_OPERATION is generated if <texture> is the name of a texture whose type does not match the texture type required by <textarget>.

The command

```
  void FramebufferTexture3DEXT(enum target, enum attachment,
                               enum textarget, uint texture,
                               int level, int zoffset);
```

behaves identically to FramebufferTextureLayerEXT, with the <layer> parameter set to the value of <zoffset>. The error INVALID_ENUM is generated if <textarget> is not TEXTURE_3D. The error INVALID_OPERATION is generated unless <texture> is zero or the name of a three-dimensional texture.

For all FramebufferTexture commands, if <texture> is non-zero and the command does not result in an error, the framebuffer attachment state corresponding to <attachment> is updated based on the new attachment. FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE_EXT is set to TEXTURE, FRAMEBUFFER_ATTACHMENT_OBJECT_NAME_EXT is set to <texture>, and FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL is set to <level>. FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_FACE is set to <textarget> if FramebufferTexture2DEXT is called and <texture> is the name of a cubemap texture; otherwise, it is set to TEXTURE_CUBE_MAP_POSITIVE_X. FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER_EXT is set to <layer> or <zoffset> if FramebufferTextureLayerEXT or FramebufferTexture3DEXT is called; otherwise, it is set to zero. FRAMEBUFFER_ATTACHMENT_LAYERED_EXT is set to TRUE if FramebufferTextureEXT is called and <texture> is the name of a three-dimensional texture, cube map texture, or one- or two-dimensional array texture; otherwise it is set to FALSE.

(modify Section 4.4.4.1, Framebuffer Attachment Completeness -- add to the conditions necessary for attachment completeness)

The framebuffer attachment point <attachment> is said to be "framebuffer attachment complete" if ...:

* If FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE_EXT is TEXTURE and FRAMEBUFFER_ATTACHMENT_OBJECT_NAME_EXT names a three-dimensional texture, FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER_EXT must be smaller than the depth of the texture.

* If FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE_EXT is TEXTURE and FRAMEBUFFER_ATTACHMENT_OBJECT_NAME_EXT names a one- or two-dimensional array texture, FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER_EXT must be smaller than the number of layers in the texture.

(modify section 4.4.4.2, Framebuffer Completeness -- add to the list of conditions necessary for completeness)

* If any framebuffer attachment is layered, all populated attachments must be layered. Additionally, all populated color attachments must be from textures of the same target (i.e., three-dimensional, cube map, or one- or two-dimensional array textures).
  { FRAMEBUFFER_INCOMPLETE_LAYER_TARGETS_EXT }

* If any framebuffer attachment is layered, all attachments must have the same layer count. For three-dimensional textures, the layer count is the depth of the attached volume. For cube map textures, the layer count is always six. For one- and two-dimensional array textures, the

```
        layer count is simply the number of layers in the array texture.
        { FRAMEBUFFER_INCOMPLETE_LAYER_COUNT_EXT }

    The enum in { brackets } after each clause of the framebuffer completeness
    rules specifies the return value of CheckFramebufferStatusEXT (see below)
    that is generated when that clause is violated. ...

    (add section 4.4.7, Layered Framebuffers)

    A framebuffer is considered to be layered if it is complete and all of its
    populated attachments are layered.  When rendering to a layered
    framebuffer, each fragment generated by the GL is assigned a layer number.
    The layer number for a fragment is zero if

      * the fragment is generated by DrawPixels, CopyPixels, or Bitmap,

      * geometry programs are disabled, or

      * the current geometry program does not contain an instruction that
        writes to the layer result binding.

    Otherwise, the layer for each point, line, or triangle emitted by the
    geometry program is taken from the layer output of the provoking vertex.
    For line strips, the provoking vertex is the second vertex of each line
    segment.  For triangle strips, the provoking vertex is the third vertex of
    each individual triangles.  The per-fragment layer can be different for
    fragments generated by each individual point, line, or triangle emitted by
    a single geometry program invocation.  A layer number written by a
    geometry program has no effect if the framebuffer is not layered.

    When fragments are written to a layered framebuffer, the fragment's layer
    number selects an image from the array of images at each attachment point
    from which to obtain the destination R, G, B, A values for blending
    (Section 4.1.8) and to which to write the final color values for that
    attachment.  If the fragment's layer number is negative or greater than
    the number of layers attached, the effects of the fragment on the
    framebuffer contents are undefined.

    When the Clear command is used to clear a layered framebuffer attachment,
    all layers of the attachment are cleared.

    When commands such as ReadPixels or CopyPixels read from a layered
    framebuffer, the image at layer zero of the selected attachment is always
    used to obtain pixel values.

    When cube map texture levels are attached to a layered framebuffer, there
    are six layers attached, numbered zero through five.  Each layer number is
    mapped to a cube map face, as indicated in Table X.4.
```

layer number	cube map face
0	TEXTURE_CUBE_MAP_POSITIVE_X
1	TEXTURE_CUBE_MAP_NEGATIVE_X
2	TEXTURE_CUBE_MAP_POSITIVE_Y
3	TEXTURE_CUBE_MAP_NEGATIVE_Y
4	TEXTURE_CUBE_MAP_POSITIVE_Z
5	TEXTURE_CUBE_MAP_NEGATIVE_Z

Table X.4, Layer numbers for cube map texture faces. The layers are numbered in the same sequence as the cube map face token values.

(modify Section 6.1.3, Enumerated Queries -- Modify/add to list of <pname> values for GetFramebufferAttachmentParameterivEXT if FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE_EXT is TEXTURE)

If <pname> is FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER_EXT and the attached image is a layer of a three-dimensional texture or one- or two-dimensional array texture, then <params> will contain the specified layer number. Otherwise, <params> will contain the value zero.

If <pname> is FRAMEBUFFER_ATTACHMENT_LAYERED_EXT, then <params> will contain TRUE if an entire level of a three-dimesional texture, cube map texture, or one- or two-dimensional array texture is attached to the <attachment>. Otherwise, <params> will contain FALSE.

(Modify the Additions to Chapter 5, section 5.4)

Add the commands FramebufferTextureEXT, FramebufferTextureLayerEXT, and FramebufferTextureFaceEXT to the list of commands that are not compiled into a display list, but executed immediately.

Dependencies on EXT_framebuffer_blit

If EXT_framebuffer_blit is supported, the EXT_framebuffer_object language should be further amended so that <target> values passed to FramebufferTextureEXT and FramebufferTextureLayerEXT can be DRAW_FRAMEBUFFER_EXT or READ_FRAMEBUFFER_EXT, and that those functions set/query state for the draw framebuffer if <target> is FRAMEBUFFER_EXT.

Dependencies on EXT_texture_array

If EXT_texture_array is not supported, the discussion array textures the layered rendering edits to EXT_framebuffer_object should be removed. Layered rendering to cube map and 3D textures would still be supported.

If EXT_texture_array is supported, the edits to EXT_framebuffer_object supersede those made in EXT_texture_array, except for language pertaining to mipmap generation of array textures.

There are no functional incompatibilities between the FBO support in these two specifications. The only differences are that this extension supports layered rendering and also rewrites certain sections of the core FBO specification more aggressively.

Dependencies on ARB_texture_rectangle

If ARB_texture_rectangle is not supported, all references to rectangle textures in the EXT_framebuffer_object spec language should be removed.

Dependencies on EXT_texture_buffer_object

If EXT_buffer_object is not supported, the reference to an INVALID_OPERATION error if a buffer texture is passed to FramebufferTextureEXT should be removed.

Dependencies on NV_primitive_restart

The spec describes the behavior that primitive restart does not affect the primitive ID counter, including for POLYGON primitives (where one could argue that the restart index starts a new primitive without a new Begin to reset the count). If NV_primitive_restart is not supported, references to that extension in the discussion of the "primitive.id" attribute should be removed.

New State

Get Value	Type	Get Command	Initial Value	Description	Sec.	Attribute
GEOMETRY_PROGRAM_NV	B	IsEnabled	FALSE	Geometry shader enable	2.14.6	enable/transform
FRAMEBUFFER_ATTACHMENT_ LAYERED_EXT	nxB	GetFramebuffer- Attachment- ParameterivEXT	FALSE	Framebuffer attachment is layered	4.4.2.3	-
GEOMETRY_VERTICES_OUT_EXT	Z+	GetProgramivARB	0	vertex limit of the current geometry program	2.14.6.4	-
GEOMETRY_INPUT_TYPE_EXT	Z+	GetProgramivARB	0	input primitive type of the current geometry program	2.14.6.4	-
GEOMETRY_OUTPUT_TYPE_EXT	Z+	GetProgramivARB	0	output primitive type of the current geometry program	2.14.6.4	-

New Implementation Dependent State

Get Value	Type	Get Command	Minimum Value	Description	Sec	Attrib
MAX_GEOMETRY_TEXTURE_ IMAGE_UNITS_EXT	Z+	GetIntegerv	16	maximum number of texture image units accessible in a geometry program	2.14.6.3	-
MAX_PROGRAM_OUTPUT_VERTICES_NV	Z+	GetProgramivARB	256	maximum number of vertices that any geometry program could emit	2.14.6.4	-
MAX_PROGRAM_TOTAL_OUTPUT_ COMPONENTS_NV	Z+	GetProgramivARB	1024	maximum number of result components (all vertices) that a geometry program can emit	2.14.6.4	-

NVIDIA Implementation Details

Because of a hardware limitation, some GeForce 8 series chips use the odd vertex of an incomplete TRIANGLE_STRIP_ADJACENCY_EXT primitive as a replacement adjacency vertex rather than ignoring it.

Issues

(1) How do geometry programs fit into the existing GL pipeline?

RESOLVED: The following diagram illustrates how geometry programs fit into the "vertex processing" portion of the GL (Chapter 2 of the OpenGL 2.0 Specification).

First, vertex attributes are specified via immediate-mode commands or through vertex arrays. They can be conventional attributes (e.g., glVertex, glColor, glTexCoord) or generic (numbered) attributes.

Vertices are then transformed, either using a vertex program or fixed-function vertex processing. Fixed-function vertex processing includes position transformation (modelview and projection matrices), lighting, texture coordinate generation, and other calculations. The results of either method are a "transformed vertex", which has a position (in clip coordinates), front and back colors, texture coordinates, generic attributes (vertex program only), and so on. Note that on many current GL implementations, vertex processing is performed by executing a "fixed function vertex program" generated by the driver.

After vertex transformation, vertices are assembled into primitives, according to the topology (e.g., TRIANGLES, QUAD_STRIP) provided by the call to glBegin(). Primitives are points, lines, triangles, quads, or polygons. Many GL implementations do not directly support quads or polygons, but instead decompose them into triangles as permitted by the spec.

After initial primitive assembly, a geometry program is executed on each individual point, line, or triangle primitive, if enabled. It can read the attributes of each transformed vertex, perform arbitrary computations, and emit new transformed vertices. These emitted vertices are themselves assembled into primitives according to the output primitive type of the geometry program.

Then, the colors of the vertices of each primitive are clamped to [0,1] (if color clamping is enabled), and flat shading may be performed by taking the color from the provoking vertex of the primitive.

Each primitive is clipped to the view volume, and to any enabled user-defined clip planes. Color, texture coordinate, and other attribute values are computed for each new vertex introduced by clipping.

After clipping, the position of each vertex (in clip coordinates) is converted to normalized device coordinates in the perspective division (divide by w) step, and to window coordinates in the viewport transformation step.

At the same time, color values may be converted to normalized fixed-point values according to the "Final Color Processing" portion of the specification.

After the vertices of the primitive are transformed to window coordinate, the GL determines if the primitive is front- or back-facing. That information is used for two-sided color selection, where a single set of colors is selected from either the front or back colors associated with each transformed vertex.

When all this is done, the final transformed position, colors (primary and secondary), and other attributes are used for rasterization (Chapter 3 in the OpenGL 2.0 Specification).

When the raster position is specified (via glRasterPos), it goes through the entire vertex processing pipeline as though it were a point. However, geometry programs are never run on the raster position.

```
                |generic              |conventional
                |vertex               |vertex
                |attributes           |attributes
                |                     |
                | +-------------------+
                | |                   |
                V V                   V
              vertex            fixed-function
              program              vertex
                 |               processing
                 |                    |
                 |                    |
                 +<-------------------+
                 |                                     Output
                 |position, color,                   Primitive
                 |other vertex data                     Type
                 |                                       |
                 V                                       |
Begin/       primitive        geometry       primitive   |
 End ------> assembly  -----> program  ----> assembly  <-+
State            |                               |
                 V                               |
                 +<------------------------------+
                 |
                 |
                 |             color            flat
                 +----------> clamping ---->  shading
                 |                               |
                 V                               |
                 +<------------------------------+
                 |
                 |
              clipping
                 |
                 |        perspective       viewport
                 +------>   divide    ----> transform
                 |                              |
                 |                          +---+-----+
                 |                          V         |
                 |           final        facing      |
                 +------>    color     determination  |
                 |         processing       |         |
                 |             |            |         |
                 |             |            |         |
                 |             +-----+ +----+         |
                 |                   | |              |
                 |                   V V              |
                 |                two-sided           |
                 |                 coloring           |
                 |                    |               |
                 |                    |               |
                 +------------------+ | +-------------+
                                    | | |
                                    V V V
                                rasterization
                                      |
                                      |
```

V

(2) Why is this called GL_NV_geometry_program4? There aren't any previous versions of this extension, let alone three?

RESOLVED: The instruction set for GPU programs of all types (vertex, fragment, and now geometry) have been unified in the GL_NV_gpu_program4 extension, and the "4" suffix in this extension name indicates the instruction set type. There are three previous NV_vertex_program variants (four if you count NV_vertex_program1_1), so "4" is the next available numeric suffix.

(3) Should the GL produce errors at Begin time if an application specifies a primitive mode that is "incompatible" with the geometry program? For example, if the geometry program operates on triangles and the application sends a POINTS primitive?

RESOLVED: Yes. Mismatches of app-specified primitive types and geometry program input primitive types seem like clear errors and would produce weird and wonderful effects.

(4) Can the input primitive type of a geometry program be changed at run time?

RESOLVED: Not in this extension. Each geometry program has a single input primitive type, and vertices are presented to the program in a specific order based on that type.

(5) Can the output primitive type of a geometry program be determined at run time?

RESOLVED: Not in this extension.

(6) Must the output primitive type of a geometry program match the input primitive type in any way?

RESOLVED: No, you can have a geometry program generate points out of triangles or triangles out of points. Some combinations are analogous to existing OpenGL operations: reading triangles and writing points or line strips can be used to emulate a subset of PolygonMode functionality. Reading points and writing triangle strips can be used to emulate point sprites.

(7) Are primitives emitted by a geometry program processed like any other OpenGL primitive?

RESOLVED: Yes. Antialiasing, stippling, polygon offset, polygon mode, culling, two-sided lighting and color selection, point sprite operations, and fragment processing all work as expected.

One limitation is that the only output primitive types supported are points, line strips, and triangle strips, none of which meaningfully support edge flags that are sometimes used in conjunction with the POINT and LINE polygon modes (edge flags are always ignored for line-mode triangle strips).

(8) Should geometry programs support additional input primitive types?

RESOLVED: Possibly in a future extension. It should be straightforward to build a future extension to support geometry programs that operate on quads. Other primitive types might be more demanding on hardware. Quads with adjacency would require 12 vertices per program execution. General polygons may require even more, since there is no fixed bound on the number of vertices in a polygon.

(9) Should geometry programs support additional output primitive types?

RESOLVED: Possibly in a future extension. Additional output types (e.g., independent lines, line loops, triangle fans, polygons) may be useful in the future; triangle fans/polygons seem particularly useful.

(10) Should we provide additional adjacency primitive types that can be used inside a Begin/End?

RESOLVED: Not in this extension. It may be desirable to add new primitive types (e.g., TRIANGLE_FAN_ADJACENCY) in a future extension.

(11) How do geometry programs interact with RasterPos?

RESOLVED: Geometry programs are ignored when specifying the raster position. While the raster position could be treated as a point, turning it into a triangle strip would be quite bizarre.

(12) How do geometry programs interact with pixel primitives (DrawPixels, Bitmap)?

RESOLVED: They do not. Fragments generated be DrawPixels and Bitmap are injected into the pipeline after the point where geometry program execution occurs.

(13) Is there a limit on the number of vertices that can be emitted by a geometry program?

RESOLVED: Unfortunately, yes. Besides practical hardware limits, there may also be practical performance advantages when applications guarantee a tight upper bound on the number of vertices a geometry shader will emit. GPUs frequently excecute programs in parallel, and there are substantial implementation challenges to parallel execution of geometry threads that can write an unbounded number of results, particular given that the all the primitives generated by the first geometry program invocation must be consumed before any of the primitives generated by the second program invocation. Limiting the amount of data a geometry program can write substantially eases the implementation burden.

A geometry program must declare a maximum number of vertices that can be emitted, called the vertex limit. There is an implementation-dependent limit on the total number of vertices a program can emit (256 minimum) and the product of the vertex limit and the number of active result components (1024 minimum). A program will fail to load if doesn't declare a limit or exceeds either of the two implementatoin-dependent limits.

It would be ideal if the limit could be inferred from the instructions in the program itself, and that would be possible for many programs, particularly ones with straight-line flow control. For programs with more complicated flow control (subroutines, data-dependent looping, and so on), it would be impossible to make such an inference and a "safe" limit would have to be used with adverse and possibly unexpected performance consequences.

The limit on the number of EMIT instructions that can be issued can not always be enforced at compile time, or even at Begin time. We specify that if a program tries to emit more vertices than the vertex limit allows, emits that exceed the limit may or may not have any effect.

(14) Should it be possible to change the limit on the number of vertices emitted by a geometry program after the program is specified?

RESOLVED: Yes, using the function ProgramVertexLimitNV(). Applications may want to tweak a piece of data that affects the number of vertices emitted, but doesn't necessarily require recompiling the entire program. Examples might be a "circular point sprite" program, that reads a single point, and draws a circle centered at that point with <N> vertices. An application could change the value <N> at run time, but it could require a change in the vertex limit. Another example might be a geometry program that does some fancy subdivision, where the relevant parameter might be a limit on how far the primitive is subdivided.

Ideally, this program object state should be set by a "program parameter" command, much like texture state is set by a "texture parameter" (TexParameter) command. Unfortunately, there are already several different "program parameter" functions:

```
ProgramEnvParameter4fARB()   -- sets global environment constants
ProgramLocalParameter4fARB() -- sets per-program constants
ProgramParameter4fNV()       -- also sets global environment constants
```

Additionally, GLSL and OpenGL 2.0 introduced "program objects" which are linked collections of vertex, fragment, and now geometry shaders. A GLSL vertex "shader" is equivalent to an ARB_vertex_program vertex "program", which is nothing like a GLSL program. As of OpenGL 2.0, GLSL programs do not have settable parameters, by subsequent extensions may want to add them (for example, EXT_geometry_shader4, which has this same functionality for GLSL). If that happens, they would want their own ProgramParameter API, but with a different prototype than this extension would want.

Naming this function "ProgramVertexLimitNV" sidesteps this issue for now.

(15) How do edge flags interact with adjacency primitives?

RESOLVED: If geometry programs are disabled, adjacency primitives are still supported. For TRIANGLES_ADJACENCY_EXT, edge flags will apply as they do for TRIANGLES. Such primitives are rendered as independent triangles as though the adjacency vertices were not provided. Edge flags for the "real" vertices are supported. For all other adjacency primitive types, edge flags are irrelevant.

(16) How do geometry programs interact with color clamping?

RESOLVED: Geometry program execution occurs prior to color clamping in the pipeline. This means the colors written by vertex programs or fixed-function vertex processing are not clamped to [0,1] before they are read by geometry programs. If color clamping is enabled, any vertex colors written by the geometry program will have their components clamped to [0,1].

(17) How are QUADS, QUAD_STRIP, and POLYGON primitives decomposed into triangles in the initial implementation of GL_NV_geometry_program4?

RESOLVED: The specification leaves the decomposition undefined, subject to a small number of rules. Assume that four vertices are specified in the order V0, V1, V2, V3.

For QUADS primitives, the quad V0->V1->V2->V3 is decomposed into the triangles V0->V1->V2, and V0->V2->V3. The provoking vertex of the quad (V3) is only found in the second triangle. If it's necessary to flat shade over an entire quad, take the attributes from V0, which will be the first vertex for both triangles in the decomposition.

For QUAD_STRIP primitives, the quad V0->V1->V3->V2 is decomposed into the triangles V0->V1->V3 and V2->V0->V3. This has the property of leaving the provoking vertex for the polygon (V3) as the third vertex for each triangle of the decomposition.

For POLYGON primitives, the polygon V0->V1->V2->V3 is decomposed into the triangles V1->V2->V0 and V2->V3->V0. This has the property of leaving the provoking vertex for the polygon (V0) as the third vertex for each triangle of the decomposition.

(18) Should geometry programs be able to select a layer of a 3D texture, cube map texture, or array texture at run time? If so, how?

RESOLVED: This extension provides a per-vertex result binding called "result.layer", which is an integer specifying the layer to render to. When an each individual point, line, or triangle is emitted by a geometry program, the layer number is taken from the provoking (last) vertex of the primitive and is used for all fragments generated by that primitive.

The EXT_framebuffer_object (FBO) extension is used for rendering to textures, but for cube maps and 3D textures, it only provides the ability to attach a single face or layer of such textures.

This extension generalizes FBO by creates new entry points to bind an entire texture level (FramebufferTextureEXT) or a single layer of a texture level (FramebufferTextureLayerEXT) to an attachment point. The existing FBO binding functions, FramebufferTexture[123]DEXT are retained, and are defined in terms of the more general new functions.

The new functions do not have a dimension in the function name or a <textarget> parameter, which can be inferred from the provided texture. They can do anything that the old functions can do, except attach a single face of a cube map texture. We considered adding a separate function FramebufferTextureFaceEXT to provide this functionality, but decided that the existing FramebufferTexture2DEXT API was adequate. We also considered using FramebufferTextureLayerEXT for this purpose, but it was not clear whether a layer number (0-5) or face enum (e.g, TEXTURE_CUBE_MAP_POSITIVE_X) should be provided.

When an entire texel level of a cube map, 3D, or array texture is attached, that attachment is considered layered. The framebuffer is considered layered if any attachment is layered. When the framebuffer is layered, there are three additional completeness requirements:

* all attachments must be layered
* all color attachments must be from textures of identical type
* all attachments must have the same number of layers

We expect subsequent versions of the FBO spec to relax the requirement that all attachments must have the same width and height, and plan to relax the similar requirement for layer count at that time.

When rendering to a layered framebuffer, layer zero is used unless a geometry program that writes the layer result is enabled. When rendering to a non-layered framebuffer, any layer result emitted from geometry programs is ignored and the set of single-image attachments are used. When reading from a layered framebuffer (e.g., ReadPixels), layer zero is always used. When clearing a layered framebuffer, all layers are cleared to the corresponding clear values.

Several other approaches were considered, including leveraging existing FBO attachment functions and requiring the use of FramebufferTexture3D with a <zoffset> of zero to make a framebuffer attachment "layerable" (attaching layer zero means that the attachment could be used for either layered- or non-layered rendering). Whether rendering was layered or not could either be inferred from the active geometry program, or set as a new property of the framebuffer object. There is presently FramebufferParameter API to set a property of a framebuffer, so it would have been necessary to create new set/query APIs if this approach were chosen.

(19) How can single-pass cube map rendering be done efficiently in a geometry program?

UNRESOLVED: To do single-pass cubemap rendering, attach entire cube map textures to framebuffer attachment points using the new functions provided by this extension. The vertex program used should only transform the vertex position to eye coordinates (position relative to the center of the cube map). A geometry program should be used that effectively projects each input triangle onto each of the six faces of the cube map, emitting a triangle for each. Each of the projected vertices should be emitted with a "result.layer" value matching the face number (0-5). When the projected triangle is drawn, it is automatically drawn on the face corresponding to the emitted layer number.

It should be simple to skip projecting primitives onto faces they won't touch. For example, if all of the X eye coordinates are positive, there is no reason to project to the "negative X" cube map face.

An example should be provided for this issue.

(20) How should per-vertex point size work with geometry programs?

RESOLVED: We will generalize the existing VERTEX_PROGRAM_POINT_SIZE enable to control the point size behavior if either vertex or geometry programs are enabled.

If geometry programs are enabled, the point size is taken from the point size result of the emitted vertex if VERTEX_PROGRAM_POINT_SIZE is enabled, or from the PointSize state otherwise.

If no geometry program is enabled, it works like OpenGL 2.0. If a vertex program is active, it's taken from the point size result or PointSize state, depending on the VERTEX_PROGRAM_POINT_SIZE enable. If no program is enabled, normal fixed-function point size handling (including distance attenuation) is supported.

This extension creates a new alias for the VERTEX_PROGRAM_POINT_SIZE enum, called PROGRAM_POINT_SIZE_EXT, to reflect that the point size enable now covers multiple program types. Both enums have the same value.

(21) How do vertex IDs work with geometry programs?

RESOLVED: Vertex IDs are automatically provided to vertex programs when applicable, via the "vertex.id" binding. However, they are not automatically copied the transformed vertex results that are read by geometry programs.

Geometry programs can read the ID of vertex <n> via the "vertex[<n>].id" binding, but the vertex ID must have been copied by the vertex program using an instruction such as:

```
MOV result.id.x, vertex.id.x;
```

If a vertex program doesn't write vertex ID, or fixed-function vertex processing is used, the vertex ID visible to geometry programs is undefined.

(22) How do primitive IDs work with geometry programs?

RESOLVED: Primitive IDs are automatically available to geometry programs via the "primitive.id" binding and indicate the number of input primitives previously processed since the last explicit or implicit Begin call.

If a geometry program wants to make the primitive ID available to a fragment program, it should copy the appropriate value to the "result.primid" binding.

(23) How do primitive IDs work with primitives not supported directly by geometry program input topologies (e.g., QUADS, POLYGON)?

RESOLVED: QUADS are decomposed into two triangles. Both triangles will have the same primitive ID, which is the number of full quads previously processed. POLYGON primitives are decomposed into a series of triangles, and all of them will have the primitive ID -- zero.

(24) This is an NV extension (NV_geometry_program4). Why do some of the new tokens have an "EXT" extension?

RESOLVED: Some of the tokens are shared between this extension and the comparable high-level GLSL programmability extension (EXT_geometry_shader4). Rather than provide a duplicate set of tokens, we simply use the EXT versions here. The tokens specific to assembly shader uses retain an NV suffix.

Revision History

None

Name

    NV_geometry_shader4

Name String

    GL_NV_geometry_shader4

Contact

    Pat Brown, NVIDIA (pbrown 'at' nvidia.com)
    Barthold Lichtenbelt, NVIDIA (blichtenbelt 'at' nvidia.com)

Status

    Shipping for GeForce 8 Series (November 2006)

Version

```
    Last Modified Date:         01/10/2007
    Author revision:            16
```

Number

    338

Dependencies

    OpenGL 1.1 is required.

    EXT_geometry_shader4 is required.

    This extension is written against the EXT_geometry_shader4 and OpenGL 2.0
    specifications.

Overview

    This extension builds upon the EXT_geometry_shader4 specification to
    provide two additional capabilities:

        * Support for QUADS, QUAD_STRIP, and POLYGON primitive types when
          geometry shaders are enabled. Such primitives will be tessellated
          into individual triangles.

        * Setting the value of GEOMETRY_VERTICES_OUT_EXT will take effect
          immediately. It is not necessary to link the program object in
          order for this change to take effect, as is the case in the EXT
          version of this extension.

New Procedures and Functions

    None

New Tokens

    None

Additions to Chapter 2 of the OpenGL 2.0 Specification (OpenGL Operation)

Modify Section 2.16.1, Geometry shader Input Primitives, of the EXT_geometry_shader4 specification as follows:

Triangles (TRIANGLES)

Geometry shaders that operate on triangles are valid for the TRIANGLES, TRIANGLE_STRIP, TRIANGLE_FAN, QUADS, QUAD_STRIP, and POLYGON primitive types.

When used with a geometry shader that operates on triangles, QUADS, QUAD_STRIP, and POLYGON primitives are decomposed into triangles in an unspecified, implementation-dependent manner. This decomposition satisfies three properties:

1. the collection of triangles fully covers the area of the original primitive,
2. no two triangles in the decomposition overlap, and
3. the orientation of each triangle is consistent with the orientation of the original primitive.

For such primitives, the shader is executed once for each triangle in the decomposition.

There are three vertices available for each program invocation. The first, second and third vertices refer to attributes of the first, second and third vertex of the triangle, respectively. ...

Modify Section 2.16.4, Geometry Shader Execution Environment, of the EXT_geometry_shader4 specification as follows:

Geometry shader inputs

(modify the spec language for primitive ID, describing its interaction with QUADS, QUAD_STRIP, and POLYGON topologies) The built-in special variable gl_PrimitiveIDIn is not an array and has no vertex shader equivalent. It is filled with the number of primitives processed since the last time Begin was called (directly or indirectly via vertex array functions). The first primitive generated after a Begin is numbered zero, and the primitive ID counter is incremented after every individual point, line, or polygon primitive is processed. For polygons drawn in point or line mode, the primitive ID counter is incremented only once, even though multiple points or lines may be drawn. For QUADS and QUAD_STRIP primitives that are decomposed into triangles, the primitive ID is incremented after each complete quad is processed. For POLYGON primitives, the primitive ID counter is undefined. Restarting a primitive topology using the primitive restart index has no effect on the primitive ID counter.

Geometry Shader outputs

(modify the vertex output limit language to allow changes to take effect immediately) A geometry shader is limited in the number of vertices it may emit per invocation. The maximum number of vertices a geometry shader can possibly emit needs to be set as a parameter of the program object that contains the geometry shader. To do so, call ProgramParameteriEXT with

<pname> set to GEOMETRY_VERTICES_OUT_EXT and <value> set to the maximum number of vertices the geometry shader will emit in one invocation. Setting this limit will take effect immediately. If a geometry shader, in one invocation, emits more vertices than the value GEOMETRY_VERTICES_OUT_EXT, these emits may have no effect.

(modify the error checking language for values that are too large) There are two implementation-dependent limits on the value of GEOMETRY_VERTICES_OUT_EXT. First, the error INVALID_VALUE will be generated by ProgramParameteriEXT if the number of vertices specified exceeds the value of MAX_GEOMETRY_OUTPUT_VERTICES_EXT. Second, the product of the total number of vertices and the sum of all components of all active varying variables may not exceed the value of MAX_GEOMETRY_TOTAL_OUTPUT_COMPONENTS_EXT. If <program> has already been successfully linked, the error INVALID_VALUE will be generated by ProgramParameteriEXT if the specified value causes this limit to be exceeded. Additionally, LinkProgram will fail if it determines that the total component limit would be violated.

Additions to Chapter 3 of the OpenGL 2.0 Specification (Rasterization)

None

Additions to Chapter 4 of the OpenGL 2.0 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the OpenGL 2.0 Specification (Special Functions)

None

Additions to Chapter 6 of the OpenGL 2.0 Specification (State and State Requests)

None

Additions to Appendix A of the OpenGL 2.0 Specification (Invariance)

None

Additions to the AGL/GLX/WGL Specifications

None

Interactions with NV_transform_feedback

If GL_NV_transform_feedback is not supported, the function GetActiveVaryingNV() needs to be added to this extension. This function can be used to count the number of varying components output by a geometry shader, and from that data the maximum value for GEOMETRY_VERTICES_OUT_EXT computed by the application.

GLX protocol

None required

Errors

The error INVALID_OPERATION is generated if Begin, or any command that implicitly calls Begin, is called when a geometry shader is active and:

* the input primitive type of the current geometry shader is POINTS and <mode> is not POINTS,

* the input primitive type of the current geometry shader is LINES and <mode> is not LINES, LINE_STRIP, or LINE_LOOP,

* the input primitive type of the current geometry shader is TRIANGLES and <mode> is not TRIANGLES, TRIANGLE_STRIP, TRIANGLE_FAN, QUADS, QUAD_STRIP, or POLYGON,

* the input primitive type of the current geometry shader is LINES_ADJACENCY_EXT and <mode> is not LINES_ADJACENCY_EXT or LINE_STRIP_ADJACENCY_EXT, or

* the input primitive type of the current geometry shader is TRIANGLES_ADJACENCY_EXT and <mode> is not TRIANGLES_ADJACENCY_EXT or TRIANGLE_STRIP_ADJACENCY_EXT.

* GEOMETRY_VERTICES_OUT_EXT is zero for the currently active program object.

New State

None

Issues

1. *Why is there a GL_NV_geometry_shader4 and a GL_EXT_geometry_shader4 extension?*

   RESOLVED: NVIDIA initially wrote the geometry shader extension, and worked with other vendors on a common extension. Most of the functionality of the original specification was retained, but a few functional changes were made, resulting in the GL_EXT_geometry_shader4 specification.

   Some of the functionality removed in this process may be useful to developers, so we chose to provide an NVIDIA extension to expose this extra functionality.

2. *Should it be possible to change the limit on the number of vertices emitted by a geometry shader after the program object, containing the shader, is linked?*

   RESOLVED: Yes. Applications may want to tweak a piece of data that affects the number of vertices emitted, but wouldn't otherwise require re-linking the entire program object. One simple example might be a "circular point sprite" shader, that reads a single point, and draws a circle centered at that point with <N> vertices, where <N> is provided as a uniform. An application could change the value <N> at run time, which would require a change in the vertex limit. Another example might be a geometry shader that does some fancy subdivision, where the relevant parameter might be a limit on how far the primitive is subdivided. This limit can be changed using the function ProgramParameteriEXT with <pname> set to GEOMETRY_VERTICES_OUT_EXT.

3. *How are QUADS, QUAD_STRIP, and POLYGON primitives decomposed into triangles in the initial implementation?*

RESOLVED: The specification leaves the decomposition undefined, subject to a small number of rules. Assume that four vertices are specified in the order V0, V1, V2, V3.

For QUADS primitives, the quad V0->V1->V2->V3 is decomposed into the triangles V0->V1->V2, and V0->V2->V3. The provoking vertex of the quad (V3) is only found in the second triangle. If it's necessary to flat shade over an entire quad, take the attributes from V0, which will be the first vertex for both triangles in the decomposition.

For QUAD_STRIP primitives, the quad V0->V1->V3->V2 is decomposed into the triangles V0->V1->V3 and V2->V0->V3. This has the property of leaving the provoking vertex for the polygon (V3) as the third vertex for each triangle of the decomposition.

For POLYGON primitives, the polygon V0->V1->V2->V3 is decomposed into the triangles V1->V2->V0 and V2->V3->V0. This has the property of leaving the provoking vertex for the polygon (V0) as the third vertex for each triangle of the decomposition.

The triangulation described here is not guaranteed to be used on all implementations of this extension, and subsequent implementations may use a more natural decomposition for QUAD_STRIP and POLYGON primitives. (For example, the triangulation of 4-vertex polygons might match that used for QUADS.)

Revision History

None

Name

    NV_gpu_program4

Name Strings

    GL_NV_gpu_program4

Contact

    Pat Brown, NVIDIA Corporation (pbrown 'at' nvidia.com)

Status

    Shipping for GeForce 8 Series (November 2006)

Version

```
    Last Modified Date:         02/04/2008
    NVIDIA Revision:            4
```

Number

    322

Dependencies

This extension is written against to OpenGL 2.0 specification.

OpenGL 2.0 is not required, but we expect all implementations of this extension will also support OpenGL 2.0.

This extension is also written against the ARB_vertex_program specification, which provides the basic mechanisms for the assembly programming model used by this extension.

This extension serves as the basis for the NV_fragment_program4, NV_geometry_program4, and NV_vertex_program4, which all build on this extension to support fragment, geometry, and vertex programs, respectively. If "GL_NV_gpu_program4" is found in the extension string, all of these extensions are supported.

NV_parameter_buffer_object affects the definition of this extension.

ARB_texture_rectangle trivially affects the definition of this extension.

EXT_gpu_program_parameters trivially affects the definition of this extension.

EXT_texture_integer trivially affects the definition of this extension.

EXT_texture_array trivially affects the definition of this extension.

EXT_texture_buffer_object trivially affects the definition of this extension.

NV_primitive_restart trivially affects the definition of this extension.

Overview

This specification documents the common instruction set and basic functionality provided by NVIDIA's 4th generation of assembly instruction sets supporting programmable graphics pipeline stages.

The instruction set builds upon the basic framework provided by the ARB_vertex_program and ARB_fragment_program extensions to expose considerably more capable hardware. In addition to new capabilities for vertex and fragment programs, this extension provides a new program type (geometry programs) further described in the NV_geometry_program4 specification.

NV_gpu_program4 provides a unified instruction set -- all instruction set features are available for all program types, except for a small number of features that make sense only for a specific program type. It provides fully capable signed and unsigned integer data types, along with a set of arithmetic, logical, and data type conversion instructions capable of operating on integers. It also provides a uniform set of structured branching constructs (if tests, loops, and subroutines) that fully support run-time condition testing.

This extension provides several new texture mapping capabilities. Shadow cube maps are supported, where cube map faces can encode depth values. Texture lookup instructions can include an immediate texel offset, which can assist in advanced filtering. New instructions are provided to fetch a single texel by address in a texture map (TXF) and query the size of a specified texture level (TXQ).

By and large, vertex and fragment programs written to ARB_vertex_program and ARB_fragment_program can be ported directly by simply changing the program header from "!!ARBvp1.0" or "!!ARBfp1.0" to "!!NVvp4.0" or "!!NVfp4.0", and then modifying the code to take advantage of the expanded feature set. There are a small number of areas where this extension is not a functional superset of previous vertex program extensions, which are documented in this specification.

New Procedures and Functions

```
    void ProgramLocalParameterI4iNV(enum target, uint index,
                                    int x, int y, int z, int w);
    void ProgramLocalParameterI4ivNV(enum target, uint index,
                                     const int *params);
    void ProgramLocalParametersI4ivNV(enum target, uint index,
                                      sizei count, const int *params);
    void ProgramLocalParameterI4uiNV(enum target, uint index,
                                     uint x, uint y, uint z, uint w);
    void ProgramLocalParameterI4uivNV(enum target, uint index,
                                      const uint *params);
    void ProgramLocalParametersI4uivNV(enum target, uint index,
                                       sizei count, const uint *params);
    void ProgramEnvParameterI4iNV(enum target, uint index,
                                  int x, int y, int z, int w);
    void ProgramEnvParameterI4ivNV(enum target, uint index,
                                   const int *params);
    void ProgramEnvParametersI4ivNV(enum target, uint index,
                                    sizei count, const int *params);
    void ProgramEnvParameterI4uiNV(enum target, uint index,
                                   uint x, uint y, uint z, uint w);
    void ProgramEnvParameterI4uivNV(enum target, uint index,
                                    const uint *params);
    void ProgramEnvParametersI4uivNV(enum target, uint index,
                                     sizei count, const uint *params);

    void GetProgramLocalParameterIivNV(enum target, uint index,
                                       int *params);
    void GetProgramLocalParameterIuivNV(enum target, uint index,
                                        uint *params);
    void GetProgramEnvParameterIivNV(enum target, uint index,
                                     int *params);
    void GetProgramEnvParameterIuivNV(enum target, uint index,
                                      uint *params);
```

New Tokens

    Accepted by the <pname> parameter of GetBooleanv, GetIntegerv,
    GetFloatv, and GetDoublev:

```
        MIN_PROGRAM_TEXEL_OFFSET_EXT                    0x8904
        MAX_PROGRAM_TEXEL_OFFSET_EXT                    0x8905
```

    (note: these tokens are shared with the EXT_gpu_shader4 extension.)

    Accepted by the <pname> parameter of GetProgramivARB:

```
        PROGRAM_ATTRIB_COMPONENTS_NV                    0x8906
        PROGRAM_RESULT_COMPONENTS_NV                    0x8907
        MAX_PROGRAM_ATTRIB_COMPONENTS_NV                0x8908
        MAX_PROGRAM_RESULT_COMPONENTS_NV                0x8909
        MAX_PROGRAM_GENERIC_ATTRIBS_NV                  0x8DA5
        MAX_PROGRAM_GENERIC_RESULTS_NV                  0x8DA6
```

Additions to Chapter 2 of the OpenGL 1.5 Specification (OpenGL Operation)

(Modify "Section 2.14.1" of the ARB_vertex_program specification, describing program parameters.)

Each program object has an associated array of program local parameters. Program local parameters are four-component vectors whose components can hold floating-point, signed integer, or unsigned integer values. The data type of each local parameter is established when the parameter's values are assigned. If a program attempts to read a local parameter using a data type other than the one used when the parameter is set, the values returned are undefined. ... The commands

```
      void ProgramLocalParameter4fARB(enum target, uint index,
                                      float x, float y, float z, float w);
      void ProgramLocalParameter4fvARB(enum target, uint index,
                                       const float *params);
      void ProgramLocalParameter4dARB(enum target, uint index,
                                      double x, double y, double z, double w);
      void ProgramLocalParameter4dvARB(enum target, uint index,
                                       const double *params);

      void ProgramLocalParameterI4iNV(enum target, uint index,
                                      int x, int y, int z, int w);
      void ProgramLocalParameterI4ivNV(enum target, uint index,
                                       const int *params);
      void ProgramLocalParameterI4uiNV(enum target, uint index,
                                       uint x, uint y, uint z, uint w);
      void ProgramLocalParameterI4uivNV(enum target, uint index,
                                        const uint *params);
```

update the values of the program local parameter numbered <index> belonging to the program object currently bound to <target>. For the non-vector versions of these commands, the four components of the parameter are updated with the values of <x>, <y>, <z>, and <w>, respectively. For the vector versions, the components of the parameter are updated with the array of four values pointed to by <params>. The error INVALID_VALUE is generated if <index> is greater than or equal to the number of program local parameters supported by <target>.

The commands

```
      void ProgramLocalParameters4fvNV(enum target, uint index,
                                       sizei count, const float *params);
      void ProgramLocalParametersI4ivNV(enum target, uint index,
                                        sizei count, const int *params);
      void ProgramLocalParametersI4uivNV(enum target, uint index,
                                         sizei count, const uint *params);
```

update the values of the program local parameters numbered <index> through <index> + <count> - 1 with the array of 4 * <count> values pointed to by <params>. The error INVALID_VALUE is generated if the sum of <index> and <count> is greater than the number of program local parameters supported by <target>.

When a program local parameter is updated, the data type of its components is assigned according to the data type of the provided values. If values provided are of type "float" or "double", the components of the parameter are floating-point. If the values provided are of type "int", the components of the parameter are signed integers. If the values provided are of type "uint", the components of the parameter are unsigned integers.

Additionally, each program target has an associated array of program environment parameters. Unlike program local parameters, program environment parameters are shared by all program objects of a given target. Program environment parameters are four-component vectors whose components can hold floating-point, signed integer, or unsigned integer values. The data type of each environment parameter is established when the parameter's values are assigned. If a program attempts to read an environment parameter using a data type other than the one used when the parameter is set, the values returned are undefined. ... The commands

```
  void ProgramEnvParameter4fARB(enum target, uint index,
                                float x, float y, float z, float w);
  void ProgramEnvParameter4fvARB(enum target, uint index,
                                 const float *params);
  void ProgramEnvParameter4dARB(enum target, uint index,
                                double x, double y, double z, double w);
  void ProgramEnvParameter4dvARB(enum target, uint index,
                                 const double *params);
  void ProgramEnvParameterI4iNV(enum target, uint index,
                                int x, int y, int z, int w);
  void ProgramEnvParameterI4ivNV(enum target, uint index,
                                 const int *params);
  void ProgramEnvParameterI4uiNV(enum target, uint index,
                                 uint x, uint y, uint z, uint w);
  void ProgramEnvParameterI4uivNV(enum target, uint index,
                                  const uint *params);
```

update the values of the program environment parameter numbered <index> for the given program target <target>. For the non-vector versions of these commands, the four components of the parameter are updated with the values of <x>, <y>, <z>, and <w>, respectively. For the vector versions, the four components of the parameter are updated with the array of four values pointed to by <params>. The error INVALID_VALUE is generated if <index> is greater than or equal to the number of program environment parameters supported by <target>.

The commands

```
  void ProgramEnvParameters4fvNV(enum target, uint index,
                                 sizei count, const float *params);
  void ProgramEnvParametersI4ivNV(enum target, uint index,
                                  sizei count, const int *params);
  void ProgramEnvParametersI4uivNV(enum target, uint index,
                                   sizei count, const uint *params);
```

update the values of the program environment parameters numbered <index> through <index> + <count> - 1 with the array of 4 * <count> values pointed to by <params>. The error INVALID_VALUE is generated if the sum of <index> and <count> is greater than the number of program local parameters supported by <target>.

When a program environment parameter is updated, the data type of its components is assigned according to the data type of the provided values. If values provided are of type "float" or "double", the components of the parameter are floating-point. If the values provided are of type "int", the components of the parameter are signed integers. If the values provided are of type "uint", the components of the parameter are unsigned integers.

Insert New Section 2.X between Sections 2.Y and 2.Z:

Section 2.X, GPU Programs

The GL provides a number of different program targets that allow an application to either replace certain fixed-function pipeline stages with a fully programmable model or use a program to control aspects of the GL pipeline that previously had only hard-wired behavior.

A common base instruction set is available for all program types, providing both integer and floating-point operations. Structured branching operations and subroutine calls are available. Texture mapping (loading data from external images) is supported for all program types. The main differences between the different program types are the set of available inputs and outputs, which are program type-specific, and a few instructions that are meaningful for only a subset of program types.

Section 2.X.2, Program Grammar

GPU program strings are specified as an array of ASCII characters containing the program text. When a GPU program is loaded by a call to ProgramStringARB, the program string is parsed into a set of tokens possibly separated by whitespace. Spaces, tabs, newlines, carriage returns, and comments are considered whitespace. Comments begin with the character "#" and are terminated by a newline, a carriage return, or the end of the program array.

The Backus-Naur Form (BNF) grammar below specifies the syntactically valid sequences for GPU programs. The set of valid tokens can be inferred from the grammar. A line containing "/* empty */" represents an empty string and is used to indicate optional rules. A program is invalid if it contains any tokens or characters not defined in this specification.

Note that this extension is not a standalone extension and a small number of grammar rules are left to be defined in the extensions defining the specific vertex, fragment, and geometry program types.

```
<program>               ::= <optionSequence> <declSequence>
                            <statementSequence> "END"

<optionSequence>        ::= <option> <optionSequence>
                          | /* empty */

<option>                ::= "OPTION" <identifier> ";"

<declSequence>          ::= /* empty */

<statementSequence>     ::= <statement> <statementSequence>
                          | /* empty */

<statement>             ::= <instruction> ";"
                          | <namingStatement> ";"
                          | <instLabel> ":"

<instruction>           ::= <ALUInstruction>
                          | <TexInstruction>
                          | <FlowInstruction>
```

```
    <ALUInstruction>        ::= <VECTORop_instruction>
                              | <SCALARop_instruction>
                              | <BINSCop_instruction>
                              | <BINop_instruction>
                              | <VECSCAop_instruction>
                              | <TRIop_instruction>
                              | <SWZop_instruction>

    <TexInstruction>        ::= <TEXop_instruction>
                              | <TXDop_instruction>

    <FlowInstruction>       ::= <BRAop_instruction>
                              | <FLOWCCop_instruction>
                              | <IFop_instruction>
                              | <REPop_instruction>
                              | <ENDFLOWop_instruction>

    <VECTORop_instruction>  ::= <VECTORop> <opModifiers> <instResult> ","
                                <instOperandV>

    <VECTORop>              ::= "ABS"
                              | "CEIL"
                              | "FLR"
                              | "FRC"
                              | "I2F"
                              | "LIT"
                              | "MOV"
                              | "NOT"
                              | "NRM"
                              | "PK2H"
                              | "PK2US"
                              | "PK4B"
                              | "PK4UB"
                              | "ROUND"
                              | "SSG"
                              | "TRUNC"

    <SCALARop_instruction>  ::= <SCALARop> <opModifiers> <instResult> ","
                                <instOperandS>

    <SCALARop>              ::= "COS"
                              | "EX2"
                              | "LG2"
                              | "RCC"
                              | "RCP"
                              | "RSQ"
                              | "SCS"
                              | "SIN"
                              | "UP2H"
                              | "UP2US"
                              | "UP4B"
                              | "UP4UB"

    <BINSCop_instruction>   ::= <BINSCop> <opModifiers> <instResult> ","
                                <instOperandS> "," <instOperandS>

    <BINSCop>               ::= "POW"
    <VECSCAop_instruction>  ::= <VECSCAop> <opModifiers> <instResult> ","
                                <instOperandV> "," <instOperandS>

    <VECSCAop>              ::= "DIV"
                              | "SHL"
                              | "SHR"
                              | "MOD"

    <BINop_instruction>     ::= <BINop> <opModifiers> <instResult> ","
                                <instOperandV> "," <instOperandV>

    <BINop>                 ::= "ADD"
                              | "AND"
                              | "DP3"
                              | "DP4"
                              | "DPH"
                              | "DST"
                              | "MAX"
                              | "MIN"
                              | "MUL"
                              | "OR"
                              | "RFL"
                              | "SEQ"
                              | "SFL"
                              | "SGE"
                              | "SGT"
                              | "SLE"
                              | "SLT"
                              | "SNE"
                              | "STR"
                              | "SUB"
                              | "XPD"
                              | "DP2"
                              | "XOR"

    <TRIop_instruction>     ::= <TRIop> <opModifiers> <instResult> ","
                                <instOperandV> "," <instOperandV> ","
                                <instOperandV>

    <TRIop>                 ::= "CMP"
                              | "DP2A"
                              | "LRP"
                              | "MAD"
                              | "SAD"
                              | "X2D"

    <SWZop_instruction>     ::= <SWZop> <opModifiers> <instResult> ","
                                <instOperandVNS> "," <extendedSwizzle>

    <SWZop>                 ::= "SWZ"

    <TEXop_instruction>     ::= <TEXop> <opModifiers> <instResult> ","
                                <instOperandV> "," <texAccess>
    <TEXop>                   ::= "TEX"
                                | "TXB"
                                | "TXF"
                                | "TXL"
                                | "TXP"
                                | "TXQ"

    <TXDop_instruction>       ::= <TXDop> <opModifiers> <instResult> ","
                                  <instOperandV> "," <instOperandV> ","
                                  <instOperandV> "," <texAccess>

    <TXDop>                   ::= "TXD"

    <BRAop_instruction>       ::= <BRAop> <opModifiers> <instTarget>
                                  <optBranchCond>

    <BRAop>                   ::= "CAL"

    <FLOWCCop_instruction>    ::= <FLOWCCop> <opModifiers> <optBranchCond>

    <FLOWCCop>                ::= "RET"
                                | "BRK"
                                | "CONT"

    <IFop_instruction>        ::= <IFop> <opModifiers> <ccTest>

    <IFop>                    ::= "IF"

    <REPop_instruction>       ::= <REPop> <opModifiers> <instOperandV>
                                | <REPop> <opModifiers>

    <REPop>                   ::= "REP"

    <ENDFLOWop_instruction> ::= <ENDFLOWop> <opModifiers>

    <ENDFLOWop>               ::= "ELSE"
                                | "ENDIF"
                                | "ENDREP"

    <opModifiers>             ::= <opModifierItem> <opModifiers>
                                | /* empty */

    <opModifierItem>          ::= "." <opModifier>

    <opModifier>              ::= "F"
                                | "U"
                                | "S"
                                | "CC"
                                | "CC0"
                                | "CC1"
                                | "SAT"
                                | "SSAT"
                                | "NTC"
                                | "S24"
                                | "U24"
                                | "HI"
      <texAccess>             ::= <texImageUnit> "," <texTarget>
                                | <texImageUnit> "," <texTarget> "," <texOffset>

      <texImageUnit>          ::= "texture" <optArrayMemAbs>

      <texTarget>             ::= "1D"
                                | "2D"
                                | "3D"
                                | "CUBE"
                                | "RECT"
                                | "SHADOW1D"
                                | "SHADOW2D"
                                | "SHADOWRECT"
                                | "ARRAY1D"
                                | "ARRAY2D"
                                | "SHADOWCUBE"
                                | "SHADOWARRAY1D"
                                | "SHADOWARRAY2D"

      <texOffset>             ::= "(" <texOffsetComp> ")"
                                | "(" <texOffsetComp> "," <texOffsetComp> ")"
                                | "(" <texOffsetComp> "," <texOffsetComp> ","
                                  <texOffsetComp> ")"

      <texOffsetComp>         ::= <optSign> <int>

      <optBranchCond>         ::= /* empty */
                                | <ccMask>

      <instOperandV>          ::= <instOperandAbsV>
                                | <instOperandBaseV>

      <instOperandAbsV>       ::= <operandAbsNeg> "|" <instOperandBaseV> "|"

      <instOperandBaseV>      ::= <operandNeg> <attribUseV>
                                | <operandNeg> <tempUseV>
                                | <operandNeg> <paramUseV>
                                | <operandNeg> <bufferUseV>

      <instOperandS>          ::= <instOperandAbsS>
                                | <instOperandBaseS>

      <instOperandAbsS>       ::= <operandAbsNeg> "|" <instOperandBaseS> "|"

      <instOperandBaseS>      ::= <operandNeg> <attribUseS>
                                | <operandNeg> <tempUseS>
                                | <operandNeg> <paramUseS>
                                | <operandNeg> <bufferUseS>

      <instOperandVNS>        ::= <attribUseVNS>
                                | <tempUseVNS>
                                | <paramUseVNS>
                                | <bufferUseVNS>

      <operandAbsNeg>         ::= <optSign>

      <operandNeg>            ::= <optSign>
      <instResult>            ::= <instResultCC>
                                | <instResultBase>

      <instResultCC>          ::= <instResultBase> <ccMask>

      <instResultBase>        ::= <tempUseW>
                                | <resultUseW>

      <namingStatement>       ::= <varMods> <ATTRIB_statement>
                                | <varMods> <PARAM_statement>
                                | <varMods> <TEMP_statement>
                                | <varMods> <OUTPUT_statement>
                                | <varMods> <BUFFER_statement>
                                | <ALIAS_statement>

      <ATTRIB_statement>      ::= "ATTRIB" <establishName> "=" <attribUseD>

      <PARAM_statement>       ::= <PARAM_singleStmt>
                                | <PARAM_multipleStmt>

      <PARAM_singleStmt>      ::= "PARAM" <establishName> <paramSingleInit>

      <PARAM_multipleStmt>    ::= "PARAM" <establishName> <optArraySize>
                                  <paramMultipleInit>

      <paramSingleInit>       ::= "=" <paramUseDB>

      <paramMultipleInit>     ::= "=" "{" <paramMultInitList> "}"

      <paramMultInitList>     ::= <paramUseDM>
                                | <paramUseDM> "," <paramMultInitList>

      <TEMP_statement>        ::= "TEMP" <varNameList>

      <OUTPUT_statement>      ::= "OUTPUT" <establishName> "=" <resultUseD>

      <varMods>               ::= <varModifier> <varMods>
                                | /* empty */

      <varModifier>           ::= "SHORT"
                                | "LONG"
                                | "INT"
                                | "UINT"
                                | "FLOAT"

      <ALIAS_statement>       ::= "ALIAS" <establishName> "=" <establishedName>

      <BUFFER_statement>      ::= <bufferDeclType> <establishName> "="
                                  <bufferSingleInit>
                                | <bufferDeclType> <establishName>
                                  <optArraySize> "=" <bufferMultInit>

      <bufferDeclType>        ::= "BUFFER"
                                | "BUFFER4"

      <bufferSingleInit>      ::= "=" <bufferUseDB>
      <bufferMultInit>        ::= "=" "{" <bufferMultInitList> "}"

      <bufferMultInitList>    ::= <bufferUseDM>
                                | <bufferUseDM> "," <bufferMultInitList>

      <varNameList>           ::= <establishName>
                                | <establishName> "," <varNameList>

      <attribUseV>            ::= <attribBasic> <swizzleSuffix>
                                | <attribVarName> <swizzleSuffix>
                                | <attribVarName> <arrayMem> <swizzleSuffix>
                                | <attribColor> <swizzleSuffix>
                                | <attribColor> "." <colorType> <swizzleSuffix>

      <attribUseS>            ::= <attribBasic> <scalarSuffix>
                                | <attribVarName> <scalarSuffix>
                                | <attribVarName> <arrayMem> <scalarSuffix>
                                | <attribColor> <scalarSuffix>
                                | <attribColor> "." <colorType> <scalarSuffix>

      <attribUseVNS>          ::= <attribBasic>
                                | <attribVarName>
                                | <attribVarName> <arrayMem>
                                | <attribColor>
                                | <attribColor> "." <colorType>

      <attribUseD>            ::= <attribBasic>
                                | <attribColor>
                                | <attribColor> "." <colorType>
                                | <attribMulti>

      <paramUseV>             ::= <paramVarName> <optArrayMem> <swizzleSuffix>
                                | <stateSingleItem> <swizzleSuffix>
                                | <programSingleItem> <swizzleSuffix>
                                | <constantVector> <swizzleSuffix>
                                | <constantScalar>

      <paramUseS>             ::= <paramVarName> <optArrayMem> <scalarSuffix>
                                | <stateSingleItem> <scalarSuffix>
                                | <programSingleItem> <scalarSuffix>
                                | <constantVector> <scalarSuffix>
                                | <constantScalar>

      <paramUseVNS>           ::= <paramVarName> <optArrayMem>
                                | <stateSingleItem>
                                | <programSingleItem>
                                | <constantVector>
                                | <constantScalar>

      <paramUseDB>            ::= <stateSingleItem>
                                | <programSingleItem>
                                | <constantVector>
                                | <signedConstantScalar>
      <paramUseDM>             ::= <stateMultipleItem>
                                | <programMultipleItem>
                                | <constantVector>
                                | <signedConstantScalar>

      <stateMultipleItem>      ::= <stateSingleItem>
                                | "state" "." <stateMatrixRows>

      <stateSingleItem>        ::= "state" "." <stateMaterialItem>
                                | "state" "." <stateLightItem>
                                | "state" "." <stateLightModelItem>
                                | "state" "." <stateLightProdItem>
                                | "state" "." <stateFogItem>
                                | "state" "." <stateMatrixRow>
                                | "state" "." <stateTexGenItem>
                                | "state" "." <stateClipPlaneItem>
                                | "state" "." <statePointItem>
                                | "state" "." <stateTexEnvItem>
                                | "state" "." <stateDepthItem>

      <stateMaterialItem>      ::= "material" "." <stateMatProperty>
                                | "material" "." <faceType> "."
                                  <stateMatProperty>

      <stateMatProperty>       ::= "ambient"
                                | "diffuse"
                                | "specular"
                                | "emission"
                                | "shininess"

      <stateLightItem>         ::= "light" <arrayMemAbs> "." <stateLightProperty>

      <stateLightProperty>     ::= "ambient"
                                | "diffuse"
                                | "specular"
                                | "position"
                                | "attenuation"
                                | "spot" "." <stateSpotProperty>
                                | "half"

      <stateSpotProperty>      ::= "direction"

      <stateLightModelItem>    ::= "lightmodel" "." <stateLModProperty>

      <stateLModProperty>      ::= "ambient"
                                | "scenecolor"
                                | <faceType> "." "scenecolor"

      <stateLightProdItem>     ::= "lightprod" <arrayMemAbs> "."
                                  <stateLProdProperty>
                                | "lightprod" <arrayMemAbs> "." <faceType> "."
                                  <stateLProdProperty>

      <stateLProdProperty>     ::= "ambient"
                                | "diffuse"
                                | "specular"
      <stateFogItem>          ::= "fog" "." <stateFogProperty>

      <stateFogProperty>      ::= "color"
                                | "params"

      <stateMatrixRows>       ::= <stateMatrixItem>
                                | <stateMatrixItem> "." <stateMatModifier>
                                | <stateMatrixItem> "." "row" <arrayRange>
                                | <stateMatrixItem> "." <stateMatModifier> "."
                                  "row" <arrayRange>

      <stateMatrixRow>        ::= <stateMatrixItem> "." "row" <arrayMemAbs>
                                | <stateMatrixItem> "." <stateMatModifier> "."
                                  "row" <arrayMemAbs>

      <stateMatrixItem>       ::= "matrix" "." <stateMatrixName>

      <stateMatModifier>      ::= "inverse"
                                | "transpose"
                                | "invtrans"

      <stateMatrixName>       ::= "modelview" <optArrayMemAbs>
                                | "projection"
                                | "mvp"
                                | "texture" <optArrayMemAbs>
                                | "program" <arrayMemAbs>

      <stateTexGenItem>       ::= "texgen" <optArrayMemAbs> "."
                                  <stateTexGenType> "." <stateTexGenCoord>

      <stateTexGenType>       ::= "eye"
                                | "object"

      <stateTexGenCoord>      ::= "s"
                                | "t"
                                | "r"
                                | "q"

      <stateClipPlaneItem>    ::= "clip" <arrayMemAbs> "." "plane"

      <statePointItem>        ::= "point" "." <statePointProperty>

      <statePointProperty>    ::= "size"
                                | "attenuation"

      <stateTexEnvItem>       ::= "texenv" <optArrayMemAbs> "."
                                  <stateTexEnvProperty>

      <stateTexEnvProperty>   ::= "color"

      <stateDepthItem>        ::= "depth" "." <stateDepthProperty>

      <stateDepthProperty>    ::= "range"

      <programSingleItem>     ::= <progEnvParam>
                                | <progLocalParam>
      <programMultipleItem>   ::= <progEnvParams>
                                | <progLocalParams>

      <progEnvParams>         ::= "program" "." "env" <arrayMemAbs>
                                | "program" "." "env" <arrayRange>

      <progEnvParam>          ::= "program" "." "env" <arrayMemAbs>

      <progLocalParams>       ::= "program" "." "local" <arrayMemAbs>
                                | "program" "." "local" <arrayRange>

      <progLocalParam>        ::= "program" "." "local" <arrayMemAbs>

      <constantVector>        ::= "{" <constantVectorList> "}"

      <constantVectorList>    ::= <signedConstantScalar>
                                | <signedConstantScalar> ","
                                  <signedConstantScalar>
                                | <signedConstantScalar> ","
                                  <signedConstantScalar> ","
                                  <signedConstantScalar>
                                | <signedConstantScalar> ","
                                  <signedConstantScalar> ","
                                  <signedConstantScalar> ","
                                  <signedConstantScalar>

      <signedConstantScalar>  ::= <optSign> <constantScalar>

      <constantScalar>        ::= <floatConstant>
                                | <intConstant>

      <floatConstant>         ::= <float>

      <intConstant>           ::= <int>

      <tempUseV>              ::= <tempVarName> <swizzleSuffix>

      <tempUseS>              ::= <tempVarName> <scalarSuffix>

      <tempUseVNS>            ::= <tempVarName>

      <tempUseW>              ::= <tempVarName> <optWriteMask>

      <resultUseW>            ::= <resultBasic> <optWriteMask>
                                | <resultVarName> <optWriteMask>

      <resultUseD>            ::= <resultBasic>

      <bufferUseV>            ::= <bufferVarName> <optArrayMem> <swizzleSuffix>

      <bufferUseS>            ::= <bufferVarName> <optArrayMem> <scalarSuffix>

      <bufferUseVNS>          ::= <bufferVarName> <optArrayMem>

      <bufferUseDB>           ::= <bufferBinding> <arrayMemAbs>
      <bufferUseDM>            ::= <bufferBinding> <arrayMemAbs>
                                 | <bufferBinding> <arrayRange>
                                 | <bufferBinding>

      <bufferBinding>          ::= "program" "." "buffer" <arrayMemAbs>

      <optArraySize>           ::= "[" "]"
                                 | "[" <int> "]"

      <optArrayMem>            ::= /* empty */
                                 | <arrayMem>

      <arrayMem>               ::= <arrayMemAbs>
                                 | <arrayMemRel>

      <optArrayMemAbs>         ::= /* empty */
                                 | <arrayMemAbs>

      <arrayMemAbs>            ::= "[" <int> "]"

      <arrayMemRel>            ::= "[" <arrayMemReg> <arrayMemOffset> "]"

      <arrayMemReg>            ::= <addrUseS>

      <arrayMemOffset>         ::= /* empty */
                                 | "+" <int>
                                 | "-" <int>

      <arrayRange>             ::= "[" <int> ".." <int> "]"

      <addrUseS>               ::= <addrVarName> <scalarSuffix>

      <ccMask>                 ::= "(" <ccTest> ")"

      <ccTest>                 ::= <ccMaskRule> <swizzleSuffix>
    <ccMaskRule>              ::= "EQ"
                                | "GE"
                                | "GT"
                                | "LE"
                                | "LT"
                                | "NE"
                                | "TR"
                                | "FL"
                                | "EQ0"
                                | "GE0"
                                | "GT0"
                                | "LE0"
                                | "LT0"
                                | "NE0"
                                | "TR0"
                                | "FL0"
                                | "EQ1"
                                | "GE1"
                                | "GT1"
                                | "LE1"
                                | "LT1"
                                | "NE1"
                                | "TR1"
                                | "FL1"
                                | "NAN"
                                | "NAN0"
                                | "NAN1"
                                | "LEG"
                                | "LEG0"
                                | "LEG1"
                                | "CF"
                                | "CF0"
                                | "CF1"
                                | "NCF"
                                | "NCF0"
                                | "NCF1"
                                | "OF"
                                | "OF0"
                                | "OF1"
                                | "NOF"
                                | "NOF0"
                                | "NOF1"
                                | "AB"
                                | "AB0"
                                | "AB1"
                                | "BLE"
                                | "BLE0"
                                | "BLE1"
                                | "SF"
                                | "SF0"
                                | "SF1"
                                | "NSF"
                                | "NSF0"
                                | "NSF1"
      <optWriteMask>          ::= /* empty */
                                | <xyzwMask>
                                | <rgbaMask>

      <xyzwMask>              ::= "." "x"
                                | "." "y"
                                | "." "xy"
                                | "." "z"
                                | "." "xz"
                                | "." "yz"
                                | "." "xyz"
                                | "." "w"
                                | "." "xw"
                                | "." "yw"
                                | "." "xyw"
                                | "." "zw"
                                | "." "xzw"
                                | "." "yzw"
                                | "." "xyzw"

      <rgbaMask>              ::= "." "r"
                                | "." "g"
                                | "." "rg"
                                | "." "b"
                                | "." "rb"
                                | "." "gb"
                                | "." "rgb"
                                | "." "a"
                                | "." "ra"
                                | "." "ga"
                                | "." "rga"
                                | "." "ba"
                                | "." "rba"
                                | "." "gba"
                                | "." "rgba"

      <swizzleSuffix>         ::= /* empty */
                                | "." <component>
                                | "." <xyzwSwizzle>
                                | "." <rgbaSwizzle>

      <extendedSwizzle>       ::= <extSwizComp> "," <extSwizComp> ","
                                  <extSwizComp> "," <extSwizComp>

      <extSwizComp>           ::= <optSign> <xyzwExtSwizSel>
                                | <optSign> <rgbaExtSwizSel>

      <xyzwExtSwizSel>        ::= "0"
                                | "1"
                                | <xyzwComponent>

      <rgbaExtSwizSel>        ::= <rgbaComponent>

      <scalarSuffix>          ::= "." <component>

      <component>             ::= <xyzwComponent>
                                | <rgbaComponent>
      <xyzwComponent>         ::= "x"
                                | "y"
                                | "z"
                                | "w"

      <rgbaComponent>         ::= "r"
                                | "g"
                                | "b"
                                | "a"

      <optSign>               ::= /* empty */
                                | "-"
                                | "+"

      <faceType>              ::= "front"
                                | "back"

      <colorType>             ::= "primary"
                                | "secondary"

      <instLabel>             ::= <identifier>

      <instTarget>            ::= <identifier>

      <establishedName>       ::= <identifier>

      <establishName>         ::= <identifier>
```

The <int> rule matches an integer constant. The integer consists of a sequence of one or more digits ("0" through "9"), or a sequence in hexadecimal form beginning with "0x" followed by a sequence of one or more hexadecimal digits ("0" through "9", "a" through "f", "A" through "F").

The <float> rule matches a floating-point constant consisting of an integer part, a decimal point, a fraction part, an "e" or "E", and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of one or more digits ("0" through "9"). Either the integer part or the fraction parts (not both) may be missing; either the decimal point or the "e" (or "E") and the exponent (not both) may be missing. Most grammar rules that allow floating-point values also allow integers matching the <int> rule.

The <identifier> rule matches a sequence of one or more letters ("A" through "Z", "a" through "z"), digits ("0" through "9), underscores ("_"), or dollar signs ("$"); the first character must not be a number. Upper and lower case letters are considered different (names are case-sensitive). The following strings are reserved keywords and may not be used as identifiers: "fragment" (for fragment programs only), "vertex" (for vertex and geometry programs), "primitive" (for fragment and geometry programs), "program", "result", "state", and "texture".

The <tempVarName>, <paramVarName>, <attribVarName>, <resultVarName>, and <bufferName> rules match identifiers that have been previously established as names of temporary, program parameter, attribute, result, and program parameter buffer variables, respectively.

The <xyzwSwizzle> and <rgbaSwizzle> rules match any 4-character strings consisting only of the characters "x", "y", "z", and "w" (<xyzwSwizzle>) or "r", "g", "b", "a" (<rgbaSwizzle>).

The error INVALID_OPERATION is generated if a program fails to load because it is not syntactically correct or for one of the semantic restrictions described in the following sections.

A successfully loaded program is parsed into a sequence of instructions. Each instruction is identified by its tokenized name. The operation of these instructions when executed is defined in section 2.X.4. A successfully loaded program string replaces the program string previously loaded into the specified program object. If the OUT_OF_MEMORY error is generated by ProgramStringARB, no change is made to the previous contents of the current program object.

Section 2.X.3, Program Variables

Programs may operate on a number of different variables during their execution. The following sections define the different classes of variables that can be declared and used by a program.

Some variable classes require variable bindings. Variable classes with bindings refer to state that is either generated or consumed outside the program. Examples of variable bindings include a vertex's normal, the position of a vertex computed by a vertex program, an interpolated texture coordinate, and the diffuse color of light 1. Variables that are used only during program execution do not have bindings.

Variables may be declared explicitly according to the <namingStatement> grammar rule. Explicit variable declarations allow a program to establish a variable name that can be used to refer to a specified resource in subsequent instructions. Variables may be declared anywhere in the program string, but must be declared prior to use. A program will fail to load if it declares the same variable name more than once, or if it refers to a variable name that has not been previously declared in the program string.

Variables may also be declared implicitly, simply by using a variable binding as an operand in a program instruction. Such uses are considered to automatically create a nameless variable using the specified binding. Only variable from classes with bindings can be declared implicitly.

Section 2.X.3.1, Program Variable Types

Explicit variable declarations may include one or more modifiers that specify additional information about the variable, such as the size and data type of the components of the variable. Variable modifiers are specified according to the <varModifier> grammar rule.

By default, variables are considered typeless. They can be used in instructions that read or write the variable as floating-point values, signed integers, or unsigned integers. If a variable is written using one data type but then read using a different one, the results of the operation are undefined. Variables with bindings are considered to be read or written when their values are produced or consumed; the data type used by the GL is specified in the description of each binding.

Explicitly declared variables may optionally have one data type modifier, which can be used to detect data type mismatch errors. Type modifers of "INT", "UINT", and "FLOAT" indicate that the components of the variable are stored as signed integers, unsigned integers, or floating-point values, respectively. A program will fail to load if it attempts to read or write a variable using a data type other than the one indicated by the data type modifier. Variables without a data type modifier can be read or written using any data type.

Explicitly declared variables may optionally have one storage size modifier. Variables decared as "SHORT" will be represented using at least 16 bits per component. "SHORT" floating-point values will have at least 5 bits of exponent and 10 bits of mantissa. Variables declared as "LONG" will be represented with at least 32 bits per component. "LONG" floating-point values will have at least 8 bits of exponent and 23 bits of mantissa. If no size modifier is provided, the GL will automatically select component sizes. Implementations are not required to support more than one component size, so "SHORT", "LONG", and the default could all refer to the same component size.

Each variable declaration can include at most one data type and one storage size modifier. A program will fail to load if it specifies multiple data type or multiple storage size modifiers in a single variable declaration.

(NOTE: Fragment programs also support the modifiers "FLAT", "CENTROID", and "NOPERSPECTIVE", which control how per-fragment attribute values are produced. These modifiers are described in detail in the NV_fragment_program4 specification.)

Explicitly declared variables of all types may be declared as arrays. An array variable has one or more members, numbered 0 through <n>-1, where <n> is the number of entries in the array. The total number of entries in the array can be declared using the <optArraySize> grammar rule. For variable classes without bindings, an array size must be specified in the program, and must be a positive integer. For variable classes with bindings, a declared size is optional, and is taken from the number of bindings assigned in the declaration if omitted. A program will fail to load if the declared size of an array variable does not match the number of assigned bindings.

When a variable is declared as an array, instructions that use the variable must specify an array member to access according to the <arrayMem> grammar rule. A program will fail to load if it contains an instruction that accesses an array variable without specifying an array member or an instruction that specifies an array member for a non-array variable.

Section 2.X.3.2, Program Attribute Variables

Program attribute variables represent per-vertex or per-fragment inputs to the program. All attribute variables have associated bindings, and are read-only during program execution. Attribute variables may be declared explicitly via the <ATTRIB_statement> grammar rule, or implicitly by using an attribute binding in an instruction.

The set of available attribute bindings depends on the program type, and is enumerated in the specifications for each program type.

The set of bindings allowed for attribute array variables is limited to attribute state grouped in arrays (e.g., texture coordinates, generic vertex attributes). Additionally, all bindings assigned to the array must be of the same binding type and must increase consecutively. Examples of valid and invalid binding lists include:

```
  vertex.attrib[1], vertex.attrib[2]      # valid, 2-entry array
  vertex.texcoord[0..3]                   # valid, 4-entry array
  vertex.attrib[1], vertex.attrib[3]      # invalid, skipped attrib 2
  vertex.attrib[2], vertex.attrib[1]      # invalid, wrong order
  vertex.attrib[1], vertex.texcoord[2]    # invalid, different types
```

Additionally, attribute bindings may be used in no more than one array variable accessed with relative addressing.

Implementations may have a limit on the total number of attribute binding components used by each program target (MAX_PROGRAM_ATTRIB_COMPONENTS). Programs that use more attribute binding components than this limit will fail to load. The method of counting used attribute binding components is implementation-dependent, but must satisfy the following properties:

* If an attribute binding is not referenced in a program, or is referenced only in declarations of attribute variables that are not used, none of its components are counted.

* An attribute binding component may be counted as used only if there exists an instruction operand where

    - the component is enabled for read by the swizzle pattern (Section 2.X.4.2), and

    - the attribute binding is

        - referenced directly by the operand,

        - bound to a declared variable referenced by the operand, or

        - bound to a declared array variable where another binding in the array satisfies one of the two previous conditions.

  Implementations are not required to optimize out unused elements of an attribute array or components that are used in only some elements of an array. The last of these rules is intended to cover the case where the same attribute binding is used in multiple variables.

  For example, an operand whose swizzle pattern selects only the x component may result in the x component of an attribute binding being counted, but may never result in the counting of the y, z, or w components of any attribute binding.

* Implementations are not required to determine that components read by an instruction are actually unused due to:

  - instruction write masks (for example, a component-wise ADD operation that only writes the "x" component doesn't have to read the "y", "z", and "w" components of its operands) or

  - any other properties of the instruction (for example, the DP3 instruction computes a 3-component dot product doesn't have to read the "w" component of its operands).

Section 2.X.3.3, Program Parameters

Program parameter variables are used as constants during program execution. All program parameter variables have associated bindings and are read-only during program execution. Program parameters retain their values across program invocations, although their values may change between invocations due to GL state changes. Program parameter variables may be declared explicitly via the <PARAM_statement> grammar rule, or implicitly by using a parameter binding in an instruction. Except where otherwise specified, program parameter bindings always specify floating-point values.

When declaring program parameter array variables, all bindings are supported and can be assigned to array members in any order. The only restriction is that no parameter binding may be used more than once in array variables accessed using relative addressing. A program will fail to load if any program parameter binding is used more than once in a single array accessed using relative addressing or used at least once in two or more arrays accessed using relative addressing.

Constant Bindings

If a program parameter binding matches the <constantScalar> or <signedConstantScalar> grammar rules, the corresponding program parameter variable is bound to the vector (X,X,X,X), where X is the value of the specified constant.

If a program parameter binding matches <constantVector>, the corresponding program parameter variable is bound to the vector (X,Y,Z,W), where X, Y, Z, and W are the values corresponding to the first, second, third, and fourth match of <signedConstantScalar>. If fewer than four constants are specified, Y, Z, and W assume the values 0, 0, and 1, if their respective constants are not specified.

Constant bindings can be interpreted as having signed integer, unsigned integer, or floating-point values, depending on how they are used in the program text. For constants in variable declarations, the components of the constant are interpreted according to the variable's component data type modifier. If no data type modifier is specified in a declaration, constants are interpreted as floating-point values. For constant bindings used directly in an instruction, the components of the constant are interpreted according to the required data type of the operand. A program will fail to load if it specifies a floating-point constant value (matching the <floatConstant> grammar rule) that should be interpreted as a signed or unsigned integer, or a negative integer constant value that should be interpreted as an unsigned integer.

If the value used to specify a floating-point constant can not be exactly represented, the nearest floating-point value will be used. If the value used to specify an integer constant is too large to be represented, the program will fail to load.

Program Environment/Local Parameter Bindings

Binding	Components	Underlying State
program.env[a]	(x,y,z,w)	program environment parameter a
program.local[a]	(x,y,z,w)	program local parameter a
program.env[a..b]	(x,y,z,w)	program environment parameters a through b
program.local[a..b]	(x,y,z,w)	program local parameters a through b

Table X.1: Program Environment/Local Parameter Bindings. <a> and <b> indicate parameter numbers, where <a> must be less than or equal to <b>.

If a program parameter binding matches "program.env[a]" or "program.local[a]", the four components of the program parameter variable are filled with the four components of program environment parameter <a> or program local parameter <a> respectively.

Additionally, for program parameter array bindings, "program.env[a..b]" and "program.local[a..b]" are equivalent to specifying program environment or local parameters <a> through <b> in order, respectively. A program using any of these bindings will fail to load if <a> is greater than <b>.

Program environment and local parameters are typeless, and may be specified as signed integer, unsigned integer, or floating-point variables. If a program environment parameter is read using a data type other than the one used to specify it, an undefined value is returned.

Material Property Bindings

Binding	Components	Underlying State
state.material.ambient	(r,g,b,a)	front ambient material color
state.material.diffuse	(r,g,b,a)	front diffuse material color
state.material.specular	(r,g,b,a)	front specular material color
state.material.emission	(r,g,b,a)	front emissive material color
state.material.shininess	(s,0,0,1)	front material shininess
state.material.front.ambient	(r,g,b,a)	front ambient material color
state.material.front.diffuse	(r,g,b,a)	front diffuse material color
state.material.front.specular	(r,g,b,a)	front specular material color
state.material.front.emission	(r,g,b,a)	front emissive material color
state.material.front.shininess	(s,0,0,1)	front material shininess
state.material.back.ambient	(r,g,b,a)	back ambient material color
state.material.back.diffuse	(r,g,b,a)	back diffuse material color
state.material.back.specular	(r,g,b,a)	back specular material color
state.material.back.emission	(r,g,b,a)	back emissive material color
state.material.back.shininess	(s,0,0,1)	back material shininess

Table X.3: Material Property Bindings. If a material face is not specified in the binding, the front property is used.

If a program parameter binding matches any of the material properties listed in Table X.3, the program parameter variable is filled according to the table. For ambient, diffuse, specular, or emissive colors, the "x", "y", "z", and "w" components are filled with the "r", "g", "b", and "a" components, respectively, of the corresponding material color. For material shininess, the "x" component is filled with the material's specular exponent, and the "y", "z", and "w" components are filled with the floating-point constants 0, 0, and 1, respectively. Bindings containing ".back" refer to the back material; all other bindings refer to the front material.

Material properties can be changed inside a Begin/End pair, either directly by calling Material, or indirectly through color material. However, such property changes are not guaranteed to update program parameter bindings until the following End command. Program parameter variables bound to material properties changed inside a Begin/End pair are undefined until the following End command.

Light Property Bindings

Binding	Components	Underlying State
state.light[n].ambient	(r,g,b,a)	light n ambient color
state.light[n].diffuse	(r,g,b,a)	light n diffuse color
state.light[n].specular	(r,g,b,a)	light n specular color
state.light[n].position	(x,y,z,w)	light n position
state.light[n].attenuation	(a,b,c,e)	light n attenuation constants and spot light exponent
state.light[n].spot.direction	(x,y,z,c)	light n spot direction and cutoff angle cosine
state.light[n].half	(x,y,z,1)	light n infinite half-angle
state.lightmodel.ambient	(r,g,b,a)	light model ambient color
state.lightmodel.scenecolor	(r,g,b,a)	light model front scene color
state.lightmodel. front.scenecolor	(r,g,b,a)	light model front scene color
state.lightmodel. back.scenecolor	(r,g,b,a)	light model back scene color
state.lightprod[n].ambient	(r,g,b,a)	light n / front material ambient color product
state.lightprod[n].diffuse	(r,g,b,a)	light n / front material diffuse color product
state.lightprod[n].specular	(r,g,b,a)	light n / front material specular color product
state.lightprod[n]. front.ambient	(r,g,b,a)	light n / front material ambient color product
state.lightprod[n]. front.diffuse	(r,g,b,a)	light n / front material diffuse color product
state.lightprod[n]. front.specular	(r,g,b,a)	light n / front material specular color product
state.lightprod[n]. back.ambient	(r,g,b,a)	light n / back material ambient color product
state.lightprod[n]. back.diffuse	(r,g,b,a)	light n / back material diffuse color product
state.lightprod[n]. back.specular	(r,g,b,a)	light n / back material specular color product

Table X.4: Light Property Bindings. <n> indicates a light number.

If a program parameter binding matches "state.light[n].ambient", "state.light[n].diffuse", or "state.light[n].specular", the "x", "y", "z", and "w" components of the program parameter variable are filled with the "r", "g", "b", and "a" components, respectively, of the corresponding light color.

If a program parameter binding matches "state.light[n].position", the "x", "y", "z", and "w" components of the program parameter variable are filled with the "x", "y", "z", and "w" components, respectively, of the light position.

If a program parameter binding matches "state.light[n].attenuation", the "x", "y", and "z" components of the program parameter variable are filled with the constant, linear, and quadratic attenuation parameters of the specified light, respectively (section 2.13.1). The "w" component of the program parameter variable is filled with the spot light exponent of the specified light.

If a program parameter binding matches "state.light[n].spot.direction", the "x", "y", and "z" components of the program parameter variable are filled with the "x", "y", and "z" components of the spot light direction of the specified light, respectively (section 2.13.1). The "w" component of the program parameter variable is filled with the cosine of the spot light cutoff angle of the specified light.

If a program parameter binding matches "state.light[n].half", the "x", "y", and "z" components of the program parameter variable are filled with the x, y, and z components, respectively, of the normalized infinite half-angle vector

```
  h_inf = || P + (0, 0, 1) ||.
```

The "w" component is filled with 1.0. In the computation of h_inf, P consists of the x, y, and z coordinates of the normalized vector from the eye position P_e to the eye-space light position P_pli (section 2.13.1). h_inf is defined to correspond to the normalized half-angle vector when using an infinite light (w coordinate of the position is zero) and an infinite viewer (v_bs is FALSE). For local lights or a local viewer, h_inf is well-defined but does not match the normalized half-angle vector, which will vary depending on the vertex position.

If a program parameter binding matches "state.lightmodel.ambient", the "x", "y", "z", and "w" components of the program parameter variable are filled with the "r", "g", "b", and "a" components of the light model ambient color, respectively.

If a program parameter binding matches "state.lightmodel.scenecolor" or "state.lightmodel.front.scenecolor", the "x", "y", and "z" components of the program parameter variable are filled with the "r", "g", and "b" components respectively of the "front scene color"

```
  c_scene = a_cs * a_cm + e_cm,
```

where a_cs is the light model ambient color, a_cm is the front ambient material color, and e_cm is the front emissive material color. The "w" component of the program parameter variable is filled with the alpha component of the front diffuse material color. If a program parameter binding matches "state.lightmodel.back.scenecolor", a similar back scene color, computed using back-facing material properties, is used. The front and back scene colors match the values that would be assigned to vertices using conventional lighting if all lights were disabled.

If a program parameter binding matches anything beginning with "state.lightprod[n]", the "x", "y", and "z" components of the program parameter variable are filled with the "r", "g", and "b" components, respectively, of the corresponding light product. The three light product components are the products of the corresponding color components of the specified material property and the light color of the specified light (see Table X.4). The "w" component of the program parameter variable is filled with the alpha component of the specified material property.

Light products depend on material properties, which can be changed inside a Begin/End pair. Such property changes are not guaranteed to take effect until the following End command. Program parameter variables bound to light products whose corresponding material property changes inside a Begin/End pair are undefined until the following End command.

Texture Coordinate Generation Property Bindings

Binding	Components	Underlying State
state.texgen[n].eye.s	(a,b,c,d)	TexGen eye linear plane coefficients, s coord, unit n
state.texgen[n].eye.t	(a,b,c,d)	TexGen eye linear plane coefficients, t coord, unit n
state.texgen[n].eye.r	(a,b,c,d)	TexGen eye linear plane coefficients, r coord, unit n
state.texgen[n].eye.q	(a,b,c,d)	TexGen eye linear plane coefficients, q coord, unit n
state.texgen[n].object.s	(a,b,c,d)	TexGen object linear plane coefficients, s coord, unit n
state.texgen[n].object.t	(a,b,c,d)	TexGen object linear plane coefficients, t coord, unit n
state.texgen[n].object.r	(a,b,c,d)	TexGen object linear plane coefficients, r coord, unit n
state.texgen[n].object.q	(a,b,c,d)	TexGen object linear plane coefficients, q coord, unit n

Table X.5: Texture Coordinate Generation Property Bindings. "[n]" is optional -- texture unit <n> is used if specified; texture unit 0 is used otherwise.

If a program parameter binding matches a set of TexGen plane coefficients, the "x", "y", "z", and "w" components of the program parameter variable are filled with the coefficients p1, p2, p3, and p4, respectively, for object linear coefficients, and the coefficents p1', p2', p3', and p4', respectively, for eye linear coefficients (section 2.10.4).

Fog Property Bindings

Binding	Components	Underlying State
state.fog.color	(r,g,b,a)	RGB fog color (section 3.10)
state.fog.params	(d,s,e,r)	fog density, linear start and end, and 1/(end-start) (section 3.10)

Table X.6: Fog Property Bindings

If a program parameter binding matches "state.fog.color", the "x", "y", "z", and "w" components of the program parameter variable are filled with the "r", "g", "b", and "a" components, respectively, of the fog color (section 3.10).

If a program parameter binding matches "state.fog.params", the "x", "y", and "z" components of the program parameter variable are filled with the fog density, linear fog start, and linear fog end parameters (section 3.10), respectively. The "w" component is filled with 1/(end-start), where end and start are the linear fog end and start parameters, respectively.

Clip Plane Property Bindings

Binding	Components	Underlying State
state.clip[n].plane	(a,b,c,d)	clip plane n coefficients

Table X.7: Clip Plane Property Bindings. <n> specifies the clip plane number, and is required.

If a program parameter binding matches "state.clip[n].plane", the "x", "y", "z", and "w" components of the program parameter variable are filled with the coefficients p1', p2', p3', and p4', respectively, of clip plane <n> (section 2.11).

Point Property Bindings

Binding	Components	Underlying State
state.point.size	(s,n,x,f)	point size, min and max size clamps, and fade threshold (section 3.3)
state.point.attenuation	(a,b,c,1)	point size attenuation consts

Table X.8: Point Property Bindings

If a program parameter binding matches "state.point.size", the "x", "y", "z", and "w" components of the program parameter variable are filled with the point size, minimum point size, maximum point size, and fade threshold, respectively (section 3.3).

If a program parameter binding matches "state.point.attenuation", the "x", "y", and "z" components of the program parameter variable are filled with the constant, linear, and quadratic point size attenuation parameters (a, b, and c), respectively (section 3.3). The "w" component is filled with 1.0.

Texture Environment Property Bindings

Binding	Components	Underlying State
state.texenv[n].color	(r,g,b,a)	texture environment n color

Table X.9: Texture Environment Property Bindings. "[n]" is optional -- texture unit <n> is used if specified; texture unit 0 is used otherwise.

If a program parameter binding matches "state.texenv[n].color", the "x", "y", "z", and "w" components of the program parameter variable are filled with the "r", "g", "b", and "a" components, respectively, of the corresponding texture environment color. Note that only "legacy" texture units, as queried by MAX_TEXTURE_UNITS, include texture environment state. Texture image units and texture coordinate sets do not have associated texture environment state.

Depth Property Bindings

Binding	Components	Underlying State
state.depth.range	(n,f,d,1)	Depth range near, far, and (far-near) (section 2.10.1)

Table X.10: Depth Property Bindings

If a program parameter binding matches "state.depth.range", the "x" and "y" components of the program parameter variable are filled with the mappings of near and far clipping planes to window coordinates, respectively. The "z" component is filled with the difference of the mappings of near and far clipping planes, far minus near. The "w" component is filled with 1.0.

Matrix Property Bindings

Binding	Underlying State
* state.matrix.modelview[n]	modelview matrix n
state.matrix.projection	projection matrix
state.matrix.mvp	modelview-projection matrix
* state.matrix.texture[n]	texture matrix n
state.matrix.program[n]	program matrix n

Table X.11: Base Matrix Property Bindings. The "[n]" syntax indicates a specific matrix number. For modelview and texture matrices, a matrix number is optional, and matrix zero will be used if the matrix number is omitted. These base bindings may further be modified by a inverse/transpose selector and a row selector.

If the beginning of a program parameter binding matches any of the matrix binding names listed in Table X.11, the binding corresponds to a 4x4 matrix. If the parameter binding is followed by ".inverse", ".transpose", or ".invtrans" (<stateMatModifier> grammar rule), the inverse, transpose, or transpose of the inverse, respectively, of the matrix specified in Table X.11 is selected. Otherwise, the matrix specified in Table X.11 is selected. If the specified matrix is poorly-conditioned (singular or nearly so), its inverse matrix is undefined. The binding name "state.matrix.mvp" refers to the product of modelview matrix zero and the projection matrix, defined as

```
 MVP = P * M0,
```

where P is the projection matrix and M0 is modelview matrix zero.

If the selected matrix is followed by ".row[<a>]" (matching the <stateMatrixRow> grammar rule), the "x", "y", "z", and "w" components of the program parameter variable are filled with the four entries of row <a> of the selected matrix. In the example,

```
PARAM m0 = state.matrix.modelview[1].row[0];
PARAM m1 = state.matrix.projection.transpose.row[3];
```

the variable "m0" is set to the first row (row 0) of modelview matrix 1 and "m1" is set to the last row (row 3) of the transpose of the projection matrix.

For program parameter array bindings, multiple rows of the selected matrix can be bound via the <stateMatrixRows> grammar rule. If the selected matrix binding is followed by ".row[<a>..<b>]", the result is equivalent to specifying matrix rows <a> through <b>, in order. A program will fail to load if <a> is greater than <b>. If no row selection is specified (<optMatrixRows> matches ""), matrix rows 0 through 3 are bound in order. In the example,

```
  PARAM m2[] = { state.matrix.program[0].row[1..2] };
  PARAM m3[] = { state.matrix.program[0].transpose };
```

the array "m2" has two entries, containing rows 1 and 2 of program matrix zero, and "m3" has four entries, containing all four rows of the transpose of program matrix zero.

Section 2.X.3.4, Program Temporaries

Program temporary variables are used to hold temporary results during program execution. Temporaries do not persist between program invocations, and are undefined at the beginning of each program invocation.

Temporary variables are declared explicitly using the <TEMP_statement> grammar rule. Each such statement can declare one or more temporaries. Temporaries can not be declared implicitly. Temporaries can be declared using any component size ("SHORT" or "LONG") and type ("FLOAT" or "INT") modifier.

Temporary variables may be declared as arrays. Temporary variables declared as arrays may be stored in slower memory than those not declared as arrays, and it is recommended to use non-array variables unless array functionality is required.

Section 2.X.3.5, Program Results

Program result variables represent the per-vertex or per-fragment results of the program. All result variables have associated bindings, are write-only during program execution, and are undefined at the beginning of each program invocation. Any vertex or fragment attributes corresponding to unwritten result variables will be undefined in subsequent stages of the pipeline. Result variables may be declared explicitly via the <OUTPUT_statement> grammar rule, or implicitly by using a result binding in an instruction.

The set of available result bindings depends on the program type, and is enumerated in the specifications for each program type.

Result variables may generally be declared as arrays, but the set of bindings allowed for arrays is limited to state grouped in arrays (e.g., texture coordinates, clip distances, colors). Additionally, all bindings assigned to the array must be of the same binding type and must increase consecutively. Examples of valid and invalid binding lists for vertex programs include:

```
  result.clip[1], result.clip[2]          # valid, 2-entry array
  result.texcoord[0..3]                  # valid, 4-entry array
  result.texcoord[1], result.texcoord[3]  # invalid, skipped texcoord 2
  result.texcoord[2], result.texcoord[1]  # invalid, wrong order
  result.texcoord[1], result.clip[2]      # invalid, different types
```

Additionally, result bindings may be used in no more than one array addressed with relative addressing.

Implementations may have a limit on the total number of result binding components used by each program target (MAX_PROGRAM_RESULT_COMPONENTS_NV). Programs that require more result binding components than this limit will fail to load. The method of counting used result binding components is implementation-dependent, but must satisfy the following properties:

* If a result binding is not referenced in a program, or is referenced only in declarations of result variables that are not used, none of its components are counted.

* A result binding component may be counted as used only if there exists an instruction operand where

    - the component is enabled in the write mask (Section 2.X.4.3), and

    - the result binding is either

        - referenced directly by the operand,

        - bound to a declared variable referenced by the operand, or

        - bound to a declared array variable where another binding in the array satisfies one of the two previous conditions.

  Implementations are not required to optimize out unused elements of an result array or components that are used in only some elements of an array. The last of these rules is intended to cover the case where the same result binding is used in multiple variables.

  For example, an instruction whose write mask selects only the x component may result in the x component of a result binding being counted, but may never result in the counting of the y, z, or w components of any result binding.

Section 2.X.3.6, Program Parameter Buffers

Program parameter buffers are arrays consisting of single-component typeless values or four-component typeless vectors stored in a buffer object. The GL provides an implementation-dependent number of buffer object binding points for each program target, to which buffer objects can be attached. Program parameter buffer variables can be changed either by updating the contents of bound buffer objects, or simply by changing the buffer object attached to a binding point.

Program parameter buffer variables are used as constants during program execution. All program parameter buffer variables have an associated binding and are read-only during program execution. Program parameter buffers retain their values across program invocations, although their values may change as buffer object bindings or contents change. Program parameter buffer variables must be declared explicitly via the <BUFFER_statement> grammar rule. Program parameter buffer bindings can not be used directly in executable instructions.

Program parameter buffer variables are treated as an array of single-component values if the <bufferDeclType> grammar rule matches "BUFFER" or as an array of four-component vectors if it matches "BUFFER4". A program will fail to load if a variable declared as "BUFFER" and another variable declared as "BUFFER4" use the same buffer binding point.

Program parameter buffer variables may be declared as arrays, but all bindings assigned to the array must use the same binding point and must increase consecutively.

Binding	Components	Underlying State
program.buffer[a][b]	(x,x,x,x)	program parameter buffer a, element b
program.buffer[a][b..c]	(x,x,x,x)	program parameter buffer a, elements b through c
program.buffer[a]	(x,x,x,x)	program parameter buffer a, all elements

Table X.12: Program Parameter Buffer Bindings. <a> indicates a buffer number, <b> and <c> indicate individual elements.

If a program parameter buffer binding matches "program.buffer[a][b]", the program parameter variable are filled with element <b> of the buffer object bound to binding point <a>. Each element of the bound buffer object is treated a one or four words of data that can hold integer or floating-point values. When a single-component binding is evaluated, the selected word is broadcast to all four components of the variable. When a four-component binding is evaluated, the four components of the buffer element are loaded into the variable. If no buffer object is bound to binding point <a>, or the bound buffer object is not large enough to hold an element <b>, the values used are undefined. The binding point <a> must be a nonnegative integer constant.

For program parameter buffer array declarations, "program.buffer[a][b..c]" is equivalent to specifying elements <b> through <c> of the buffer object bound to binding point <a> in order.

For program parameter buffer array declarations, "program.buffer[a]" is equivalent to specifying the entire buffer -- elements 0 through <n>-1, where <n> is either the size of the array (if declared) or the implementation-dependent maximum parameter buffer object size limit (if no size is declared).

Section 2.X.3.7, Program Condition Code Registers

The program condition code registers are four-component vectors. Each component of this register is a collection of single-bit flags, including a sign flag (SF), a zero flag (ZF), an overflow flag (OF), and a carry flag (CF). There are two condition code registers (CC0 and CC1), whose values are undefined at the beginning of program execution.

Most program instructions can optionally update one of the condition code registers, by designating the condition code to update in the instruction. When a condition code component is updated, the four flags of each component of the condition code are set according to the corresponding component of the instruction result. Full details on the condition code updates and tests can be found in Section 2.X.4.3.

The value of these four flags can be combined in various condition code tests, which can be used to mask writes to destination variables and to perform conditional branches or other condition operations.

Section 2.X.3.8, Program Aliases

Programs can create aliases by matching the <ALIAS_statement> grammar rule. Aliases allow programs to use multiple variable names to refer to a single underlying variable. For example, the statement

```
  ALIAS var1 = var0
```

establishes a variable name of "var1". Subsequent references to "var1" in the program text are treated as references to "var0". The left hand side of an ALIAS statement must be a new variable name, and the right hand side must be an established variable name.

Aliases are not considered variable declarations, so do not count against the limits on the number of variable declarations allowed in the program text.

Section 2.X.3.9, Program Resource Limits

(see ARB_vertex_program specification, incorporates all the different limits on instruction counts, temporaries, attribute bindings, program parameters, and so on)

Section 2.X.4, Program Execution Environment

The set of instructions supported for GPU programs is given in Table X.13 below and is described in detail in Section 2.X.8. An instruction can use up to three operands when it executes, and most instructions can write a single result vector. Instructions may also specify one or more modifiers, according to the <opModifiers> grammar rule. Instruction modifiers affect how the specified operation is performed.

GPU programs may operate on signed integer, unsigned integer, or floating-point values; some instructions are capable of operating on any of the three types. However, the data type of the operands and the result are always determined based solely on the instruction and its modifiers. If any of the variables used in the instruction are typeless, they will be interpreted according to the data type derived from the instruction. If any variables with a conflicting data type are used in the instruction, the program will fail to load unless the "NTC" (no type checking) instruction modifier is specified.

	Modifiers								
Instruction	F	I	C	S	H	D	Out	Inputs	Description
ABS	X	X	X	X	X	F	v	v	absolute value
ADD	X	X	X	X	X	F	v	v,v	add
AND	-	X	X	-	-	S	v	v,v	bitwise and
BRK	-	-	-	-	-	-	-	c	break out of loop instruction
CAL	-	-	-	-	-	-	-	c	subroutine call
CEIL	X	X	X	X	X	F	v	vf	ceiling
CMP	X	X	X	X	X	F	v	v,v,v	compare
CONT	-	-	-	-	-	-	-	c	continue with next loop interation
COS	X	-	X	X	X	F	s	s	cosine with reduction to [-PI,PI]
DIV	X	X	X	X	X	F	v	v,s	divide vector components by scalar
DP2	X	-	X	X	X	F	s	v,v	2-component dot product
DP2A	X	-	X	X	X	F	s	v,v,v	2-comp. dot product w/scalar add
DP3	X	-	X	X	X	F	s	v,v	3-component dot product
DP4	X	-	X	X	X	F	s	v,v	4-component dot product
DPH	X	-	X	X	X	F	s	v,v	homogeneous dot product
DST	X	-	X	X	X	F	v	v,v	distance vector
ELSE	-	-	-	-	-	-	-	-	start if test else block
ENDIF	-	-	-	-	-	-	-	-	end if test block
ENDREP	-	-	-	-	-	-	-	-	end of repeat block
EX2	X	-	X	X	X	F	s	s	exponential base 2
FLR	X	X	X	X	X	F	v	vf	floor
FRC	X	-	X	X	X	F	v	v	fraction
I2F	-	X	X	-	-	S	vf	v	integer to float
IF	-	-	-	-	-	-	-	c	start of if test block
KIL	X	X	-	-	X	F	-	vc	kill fragment
LG2	X	-	X	X	X	F	s	s	logarithm base 2
LIT	X	-	X	X	X	F	v	v	compute lighting coefficients
LRP	X	-	X	X	X	F	v	v,v,v	linear interpolation
MAD	X	X	X	X	X	F	v	v,v,v	multiply and add
MAX	X	X	X	X	X	F	v	v,v	maximum
MIN	X	X	X	X	X	F	v	v,v	minimum
MOD	-	X	X	-	-	S	v	v,v	modulus vector components by scalar
MOV	X	X	X	X	X	F	v	v	move
MUL	X	X	X	X	X	F	v	v,v	multiply
NOT	-	X	X	-	-	S	v	v	bitwise not
NRM	X	-	X	X	X	F	v	v	normalize 3-component vector
OR	-	X	X	-	-	S	v	v,v	bitwise or
PK2H	X	X	-	-	-	F	s	vf	pack two 16-bit floats
PK2US	X	X	-	-	-	F	s	vf	pack two floats as unsigned 16-bit
PK4B	X	X	-	-	-	F	s	vf	pack four floats as signed 8-bit
PK4UB	X	X	-	-	-	F	s	vf	pack four floats as unsigned 8-bit
POW	X	-	X	X	X	F	s	s,s	exponentiate
RCC	X	-	X	X	X	F	s	s	reciprocal (clamped)
RCP	X	-	X	X	X	F	s	s	reciprocal
REP	X	X	-	-	X	F	-	v	start of repeat block
RET	-	-	-	-	-	-	-	c	subroutine return
RFL	X	-	X	X	X	F	v	v,v	reflection vector
ROUND	X	X	X	X	X	F	v	vf	round to nearest integer
RSQ	X	-	X	X	X	F	s	s	reciprocal square root
SAD	-	X	X	-	-	S	vu	v,v,vu	sum of absolute differences
SCS	X	-	X	X	X	F	v	s	sine/cosine without reduction
SEQ	X	X	X	X	X	F	v	v,v	set on equal
SFL	X	X	X	X	X	F	v	v,v	set on false
SGE	X	X	X	X	X	F	v	v,v	set on greater than or equal

```
              Modifiers
Instruction F I C S H D  Out Inputs    Description
----------- - - - - - -  --- --------  -------------------------------
SGT         X X X X X F  v   v,v       set on greater than
SHL         - X X - - S  v   v,s       shift left
SHR         - X X - - S  v   v,s       shift right
SIN         X - X X X F  s   s         sine with reduction to [-PI,PI]
SLE         X X X X X F  v   v,v       set on less than or equal
SLT         X X X X X F  v   v,v       set on less than
SNE         X X X X X F  v   v,v       set on not equal
SSG         X - X X X F  v   v         set sign
STR         X X X X X F  v   v,v       set on true
SUB         X X X X X F  v   v,v       subtract
SWZ         X - X X X F  v   v         extended swizzle
TEX         X X X X - F  v   vf        texture sample
TRUNC       X X X X X F  v   vf        truncate (round toward zero)
TXB         X X X X - F  v   vf        texture sample with bias
TXD         X X X X - F  v   vf,vf,vf  texture sample w/partials
TXF         X X X X - F  v   vs        texel fetch
TXL         X X X X - F  v   vf        texture sample w/LOD
TXP         X X X X - F  v   vf        texture sample w/projection
TXQ         - - - - - S  vs  vs        texture info query
UP2H        X X X X - F  vf  s         unpack two 16-bit floats
UP2US       X X X X - F  vf  s         unpack two unsigned 16-bit ints
UP4B        X X X X - F  vf  s         unpack four signed 8-bit ints
UP4UB       X X X X - F  vf  s         unpack four unsigned 8-bit ints
X2D         X - X X X F  v   v,v,v     2D coordinate transformation
XOR         - X X - - S  v   v,v       exclusive or
XPD         X - X X X F  v   v,v       cross product
```

Table X.13: Summary of NV_gpu_program4 instructions. The "Modifiers" columns specify the set of modifiers allowed for the instruction:

```
  F = floating-point data type modifiers
  I = signed and unsigned integer data type modifiers
  C = condition code update modifiers
  S = clamping (saturation) modifiers
  H = half-precision float data type suffix
  D = default data type modifier (F, U, or S)
```

The input and output columns describe the formats of the operands and results of the instruction.

```
  v:  4-component vector (data type is inherited from operation)
  vf: 4-component vector (data type is always floating-point)
  vs: 4-component vector (data type is always signed integer)
  vu: 4-component vector (data type is always unsigned integer)
  s:  scalar (replicated if written to a vector destination;
              data type is inherited from operation)
  c:  condition code test result (e.g., "EQ", "GT1.x")
  vc: 4-component vector or condition code test
```

Section 2.X.4.1, Program Instruction Modifiers

There are several types of instruction modifiers available. A data type modifier specifies that an instruction should operate on signed integer, unsigned integer, or floating-point data, when multiple data types are supported. A clamping modifier applies to instructions with floating-point results, and specifies the range to which the results should be clamped. A condition code update modifier specifies that the instruction should update one of the condition code variables. Several other special modifiers are also provided.

Instruction modifiers may be specified as stand-alone modifiers or as suffixes concatenated with the opcode name. A program will fail to load if it contains an instruction that

* specifies more than one modifier of any given type,

* specifies a clamping modifier on an instruction, unless it produces floating-point results, or

* specifies a modifier that is not supported by the instruction (see Table X.13 and the instruction description).

Stand-alone instruction modifiers are specified according to the <opModifiers> grammar rule using a ".<modifier>" syntax. Multiple modifers, separated by periods, may be specified. The set of supported modifiers is described in Table X.14.

Modifier	Description
F	Floating-point operation
U	Fixed-point operation, unsigned operands
S	Fixed-point operation, signed operands
CC	Update condition code register zero
CC0	Update condition code register zero
CC1	Update condition code register one
SAT	Floating-point results clamped to [0,1]
SSAT	Floating-point results clamped to [-1,1]
NTC	Disable type-checking on operands/results
S24	Signed multiply (24-bit operands)
U24	Unsigned multiply (24-bit operands)
HI	Multiplies two 32-bit integer operands, returns the 32 MSBs of the product

Table X.14, Instruction Modifers.

"F", "U", and "S" modifiers are data type modifiers and specify that the instruction should operate on floating-point, unsigned integer, or signed integer values, respectively. For example, "ADD.F", "ADD.U", and "ADD.S" specify component-wise addition of floating-point, unsigned integer, or signed integer vectors, respectively. These modifiers specify a data type, but do not specify a precision at which the operation is performed. Floating-point operations will be carried out with an internal precision no less than that used to represent the largest operand. Fixed-point operations will be carried out using at least as many bits as used to represent the largest operand. Operands represented with fewer bits than used to perform the instruction will be promoted to a larger data type. Signed integer operands will be sign-extended, where the most significant bits are filled with ones if the operand is negative and zero otherwise. Unsigned integer operands will be zero-extended, where the most significant bits are always filled with zeroes. For some instructions, the data type of some operands or the result are fixed; in these cases, the data type modifier specifies the data type of the remaining values.

"CC", "CC0", and "CC1" are condition code update modifiers that specify that one of the condition code registers should be updated based on the result of the instruction, as described in section 2.X.4.3. "CC" and "CC0" specify that the condition code register CC0 be updated; "CC1" specifies an update to CC1. If no condition code update modifier is provided, the condition code registers will not be affected.

"SAT" and "SSAT" are clamping modifiers that specify that the floating-point components of the instruction result should be clamped to [0,1] or [-1,1], respectively, before updating the condition code and the destination variable. If no clamping suffix is specified, unclamped results will be used for condition code updates (if any) and destination variable writes. Clamping modifiers are not supported on instructions that do not produce floating-point results.

"NTC" (no type checking) disables data type checking on the instruction, and allows instructions to use operands or result variables whose data types are inconsistent with the expected data types of the instruction.

"S24", "U24", and "HI" are special modifiers that are allowed only for the MUL instruction, and are described in detail where MUL is documented. No more than one such modifier may be provided for any instruction.

If an instruction supports data type modifiers, but none is provided, a default data type will be chosen based on the instruction, as specified in Table X.13 and the instruction set description (Section 2.X.8). If condition code update or clamping modifiers are not specified, the corresponding operation will not be performed.

Additionally, each instruction name may have one or more suffixes, concatenated onto the base instruction name, that operate as instruction modifiers. For conciseness, these suffixes are not spelled out in the grammar -- the base opcode name is used as a placeholder for the opcode and all of its possible suffixes. Instruction suffixes are provided mainly for compatibility with prior GPU program instruction sets (e.g., NV_vertex_program3, NV_fragment_program2, and predecessors). The set of allowable suffixes, and their equivalent stand-alone modifiers, are listed in Table X.15.

Suffix	Modifier	Description
R	F	Floating-point operation, 32-bit precision
H	F(*)	Floating-point operation, at least 16-bit precision
C	CC0	Update condition code register zero
C0	CC0	Update condition code register zero
C1	CC1	Update condition code register one
_SAT	SAT	Floating-point results clamped to [0,1]
_SSAT	SSAT	Floating-point results clamped to [-1,1]

Table X.15, Instruction Suffixes.

The "R" and "H" suffixes specify floating-point operations and are equivalent to the "F" data type modifier. They additionally specify a minimum precision for the operations. Instructions with an "R" precision modifier will be carried out at no less than IEEE single-precision floating-point (8 bits of exponent, 23 bits of mantissa). Instructions with an "H" precision modifier will be carried out at no less than 16-bit floating-point precision (5 bits of exponent, 10 bits of mantissa).

An instruction may have multiple suffixes, but they must appear in order, with data type suffixes first, followed by condition code update suffixes, followed by clamping suffixes. For example, "ADDR" carries out an add at 32-bit precision. "ADDH_SAT" carries out an add at 16-bit precision (or better) and clamps the results to [0,1]. "ADDRC1_SSAT" carries out an add at 32-bit floating-point precision, clamps the results to [-1,1], and updates condition code one based on the clamped result.

Section 2.X.4.2, Program Operands

Most program instructions operate on one or more scalar or vector operands. Each operand specifies an operand variable, which is either the name of a previously declared variable or an implicit variable declaration created by using a variable binding in the instruction. Attribute, parameter, or parameter buffer variables can be declared implicitly by using a valid binding name in an operand. Instruction operands are specified by the <instOperandV>, <instOperandS>, or <instOperandVNS> grammar rules.

If the operand variable is not an array, its contents are loaded directly. If the operand variable is an array, a single element of the array is loaded according to the <arrayMem> grammar rule. The elements of an array are numbered from 0 to <n>-1, where <n> is the number of entries in the array. Array members can be accessed using either absolute or relative addressing.

Absolute array addressing is used when the <arrayMemAbs> grammar rule is matched; the array member to load is specified by the matching integer. Out-of-bounds array absolute accesses are not allowed. If the specified member number is greater than or equal to the size of the array, the program will fail to load.

Relative array addressing is used when the <arrayMemRel> grammar rule is matched. This grammar rule allows the program to specify a scalar integer operand and an optional constant offset, according to the <arrayMemReg> and <arrayMemOffset> grammar rules. When performing relative addressing, the GL evaluates the specified integer scalar operand (according to the rules specified in this section) and adds the constant offset. The array member loaded is given by this sum. The constant offset is considered zero if an offset is omitted. If the sum is negative or exceeds the size of the array, the results of the access are undefined, but may not lead to program or GL termination. The set of constant offsets supported for relative addressing is limited to values in the range [0,<n>-1], where <n> is the size of the array. A program will fail to load if it specifies an offset outside that range. If offsets outside that range are required, they can be applied by using an integer ADD instruction writing to a temporary variable.

After the operand is loaded, its components can be rearranged according to the <swizzleSuffix> grammar rule, or it can be converted to a scalar operand according to the <scalarSuffix> grammar rule.

The <swizzleSuffix> grammar rule rearranges the components of a loaded vector to produce another vector. If the <swizzleSuffix> rule matches the <xyzwSwizzle> or <rgbaSwizzle> grammar rule, a pattern of the form ".????" is used, where each question mark is replaced with one of "x", "y", "z", "w", "r", "g", "b", or a". For such patterns, the x, y, z, and w components of the operand are taken from the vector components named by the first, second, third, and fourth character of the pattern, respectively. Swizzle components of "r", "g", "b", and "a" are equivalent to "x", "y", "z", and "w", respectively. For example, if the swizzle suffix is ".yzzx" or ".gbbr" and the specified source contains {2,8,9,0}, the result is the vector {8,9,9,2}. If the <swizzleSuffix> matches the <component> grammar rule, a pattern of the form ".?" is used. For this pattern, all four components of the operand are taken from the single component identified by the pattern. If the swizzle suffix is omitted, components are not rearranged and swizzling has no effect, as though ".xyzw" were specified.

The swizzle suffix rules do not allow mixing "x", "y", "z", or "w" selectors with "r", "g", "b", or "a" selectors. A program will fail to load if it contains a swizzle suffix with selectors from both of these sets.

The <scalarSuffix> grammar rule converts a vector to a scalar by selecting a single component. The <scalarSuffix> rule is similar to the swizzle selector, except that only a single component is selected. If the scalar suffix is ".y" and the specified source contains {2,8,9,0}, the value is the scalar value 8.

Next, a component-wise negate operation is performed on the operand if the <operandNeg> grammar rule matches "-". Negation is not performed if the operand has no sign prefix, or is prefixed with "+". For unsigned integer operands, the negate operand performs a two's complement operation.

Next, a component-wise absolute value operation is performed on the operand if the <instOperandAbsV> or <instOperandAbsS> grammar rule is matched, by surrounding the operand with two "|" characters. The result is optionally negated if the <operandAbsNeg> grammar rule matches "-". For unsigned integer operands, the absolute value operation has no effect.

Section 2.X.4.3, Program Destination Variable Update

Most program instructions perform computations that produce a result, which will be written to a variable. Each instruction that computes a result specifies a destination variable, which is either the name of a previously declared variable or an implicit variable declaration created by using a variable binding in the instruction. Result variables can be declared implicitly by using a valid program result binding name in the result portion of the instruction. Instruction results are specified according to the <instResult> grammar rule.

The destination variable may be a single member of an array. In this case, a single array member is specified using the <arrayMem> grammar rule, and the array member to update is computed in the exact same manner as done for operand loads. If the array member is computed at run time, and is negative or greater than or equal to the size of the array, the results of the destination variable update are undefined and could result in overwriting other program variables.

The results of the operation may be obtained at a different precision than that used to store the destination variable. If so, the results are converted to match the size of the destination variable. For floating-point values, the results are rounded to the nearest floating-point value that can be represented in the destination variable. If a result component is larger in magnitude than the largest representable floating-point value in the data type of the destination variable, an infinity encoding (+/-INF) is used. Signed or unsigned integer values are sign-extended or zero-extended, respectively, if the destination variable has more bits than the result, and have their most significant bits discarded if the destination variable has fewer bits.

Writes to individual components of a vector destination variable can be controlled at compile time by individual component write masks specified in the instruction. The component write mask is specified by the <optWriteMask> grammar rule, and is a string of up to four characters, naming the components to enable for writing. If no write mask is specified, all components are enabled for writing. The characters "x", "y", "z", and "w" match the x, y, z, and w components respectively. For example, a write mask mask of ".xzw" indicates that the x, z, and w components should be enabled for writing but the y component should not be written. The grammar requires that the destination register mask components must be listed in "xyzw" order. Additionally, write mask components of "r", "g", "b", and "a" are equivalent to "x", "y", "z", and "w", respectively. The grammar does not allow mixing "x", "y", "z", or "w" components with "r", "g", "b", and "a" ones.

Writes to individual components of a vector destination variable, or to a scalar destination variable, can also be controlled at run time using condition code write masks. The condition code write mask is specified by the <ccMask> grammar rule. If a mask is specified, a condition code variable is loaded according to the <ccMaskRule> grammar rule and tested as described in Table X.16 to produce a four-component vector of TRUE/FALSE values.

mask rule	test name	condition
EQ, EQ0, EQ1	equal	!SF && ZF
GE, GE0, GE1	greater than or equal	!(SF ^ OF)
GT, GT0, GT1	greater than	(!SF ^ OF) && !ZF
LE, LE0, LE1	less than or equal	SF ^ (ZF || OF)
LT, LT0, LT1	less than	(SF && !ZF) ^ OF
NE, NE0, NE1	not equal	SF || !ZF
FL, FL0, FL1	false	always false
TR, TR0, TR1	true	always true
NAN, NAN0, NAN1	not a number	SF && ZF
LEG, LEG0, LEG1	less, equal, or greater (anything but a NaN)	!SF || !ZF
CF, CF0, CF1	carry flag	CF
NCF, NCF0, NCF1	no carry flag	!CF
OF, OF0, OF1	overflow flag	OF
NOF, NOF0, NOF1	no overflow flag	!OF
SF, SF0, SF1	sign flag	SF
NSF, NSF0, NSF1	no sign flag	!SF
AB, AB0, AB1	above	CF && !ZF
BLE, BLE0, BLE1	below or equal	!CF || ZF

Table X.16, Condition Code Tests. The allowed rules are specified in the "mask rule" column. If "0" or "1" is appended to the rule name (e.g., "EQ1"), the corresponding condition code register (CC1 in this example) is loaded, otherwise CC0 is loaded. After loading, each component is tested, using the expression listed in the "condition" column.

After the condition code tests are performed, the four-component result can be swizzled according to the <swizzleSuffix> grammar rule. Individual components of the destination variable are written only if the corresponding component of the swizzled condition code test result is TRUE. If both a (compile-time) component write mask and a condition code write mask are specified, destination variable components are written only if the corresponding component is enabled in both masks.

A program instruction can also optionally update one of the two condition code registers if the "CC", "CC0", or "CC1" instruction modifier are specified. These instruction modifiers update condition code register CC0, CC0, or CC1, respectively. The instructions "ADD.CC" or "ADD.CC0" will perform an add and update condition code zero, "ADD.CC1" will add and update condition code one, and "ADD" will simply perform the add without a condition code update. The components of the selected condition code register are updated if and only if the corresponding component of the destination variable are enabled by both write masks. For the purposes of condition code update, a scalar destination variable is treated as a vector where the scalar result is written to "x" (if enabled in the write mask), and writes to the "y", "z", and "w" components are disabled.

When condition code components are written, the condition code flags are updated based on the corresponding component of the result. If a component of the destination register is not enabled for writes, the corresponding condition code component is also unchanged.

For floating-point results, the sign flag (SF) is set if the result is less than zero or is a NaN (not a number) value. The zero flag (ZF) is set if the result is equal to zero or is a NaN.

For signed and unsigned integer results, the sign flag (SF) is set if the most significant bit of the value written to the result variable is set and the zero flag (ZF) is set if the result written is zero. For instructions other than those performing an integer add or subtract (ADD, MAD, SAD, SUB), the overflow and carry flags (OF and CF) are cleared.

For integer add or subtract operations, the overflow and carry flags by doing both signed and unsigned adds/subtracts as follows:

The overflow flag (OF) is set by interpreting the two operands as signed integers and performing a signed add or subtract. If the result is representable as a signed integer (i.e., doesn't overflow), the overflow flag is cleared; otherwise, it is set.

The carry flag (CF) is set by interpreting the two operands as unsigned integers and performing an unsigned add or subtract. If the result of an add is representable as an unsigned integer (i.e., doesn't overflow), the carry flag is cleared; otherwise, it is set. If the result of a subtract is greater than or equal to zero, the carry flag is set; otherwise, it is cleared.

For the purposes of condition code setting, negation modifiers turn add operations into subtracts and vice versa. If the operation is equivalent to an add with both operands negated (-A-B), the carry and overflow flags are both undefined.

Section 2.X.4.4, Program Texture Access

Certain program instructions may access texture images, as described in section 3.8. The coordinates, level-of-detail, and partial derivatives used for performing the texture lookup are derived from values provided in the program as described in the various sub-sections of Section 2.X.8. These descriptions use the function

```
  result_t_vec
    TextureSample(float_vec coord, float lod, float_vec ddx,
                  float_vec ddy, int_vec offset);
```

which obtains a filtered texel value <tau> as described in Section 3.8.8 and returns a 4-component vector (R,G,B,A) according to the format conversions specified in Table 3.21. The result vector is interpreted as floating-point, signed integer, or unsigned integer, according to the data type modifier of the instruction. If the internal format of the texture does not match the instruction's data type modifer, the results of the texture lookup are undefined.

(Note: For unextended OpenGL 2.0, all supported texture internal formats store integer values but return floating-point results in the range [0,1] on a texture lookup. The ARB_texture_float extension introduces floating-point internal format where components are both stored and returned as floating-point values. The EXT_texture_integer extension introduces formats that both store and return either signed or unsigned integer values.)

<coord> is a four-component floating-point vector from which the (s,t,r) texture coordinates used for the texture access, the layer used for array textures, and the reference value used for depth comparisons (section 3.8.14) are extracted according to Table X.17. If the texture is a cube map, (s,t,r) is projected to one of the six cube faces to produce a new (s,t) vector according to Section 3.8.6. For array textures, the layer used is derived by rounding the extracted floating-point component to the nearest integer and clamping the result to the range [0,<n>-1], where <n> is the number of layers in the texture.

<lod> specifies the level of detail parameter and replaces the value computed in equation 3.18. <ddx> and <ddy> specify partial derivatives (ds/dx, dt/dx, dr/dx, ds/dy, dt/dy, and dr/dy) for the texture coordinates, and may be used to derive footprint shapes for anisotropic texture filtering.

<offset> is a constant 3-component signed integer vector specified according to the <texOffset> grammar rule, which is added to the computed <u>, <v>, and <w> texel locations prior to sampling. One, two, or three components may be specified in the instruction; if fewer than three are specified, the remaining offset components are zero. A limited range of offset values are supported; the minimum and maximum <texOffset> values are implementation-dependent and given by MIN_PROGRAM_TEXEL_OFFSET_EXT and MAX_PROGRAM_TEXEL_OFFSET_EXT, respectively. A program will fail to load:

* if the texture target specified in the instruction is 1D, ARRAY1D, SHADOW1D, or SHADOWARRAY1D, and the second or third component of the offset vector is non-zero,

* if the texture target specified in the instruction is 2D, RECT, ARRAY2D, SHADOW2D, SHADOWRECT, or SHADOWARRAY2D, and the third component of the offset vector is non-zero,

* if the texture target is CUBE or SHADOWCUBE, and any component of the offset vector is non-zero -- texel offsets are not supported for cube map or buffer textures, or

* if any component of the offset vector is less than MIN_PROGRAM_TEXEL_OFFSET_EXT or greater than MAX_PROGRAM_TEXEL_OFFSET_EXT.

(NOTE: Texel offsets are a new feature provided by this extension and are described in more detail in edits to Section 3.8 below.)

The texture used by TextureSample() is one of the textures bound to the texture image unit whose number is specified in the instruction according to the <texImageUnit> grammar rule. The texture target accessed is specified according to the <texTarget> grammar rule and Table X.17. Fixed-function texture enables are always ignored when determining the texture to access in a program.

texTarget	Texture Type	coordinates used s t r	coordinates used layer	coordinates used shadow
1D	TEXTURE_1D	x - -	-	-
2D	TEXTURE_2D	x y -	-	-
3D	TEXTURE_3D	x y z	-	-
CUBE	TEXTURE_CUBE_MAP	x y z	-	-
RECT	TEXTURE_RECTANGLE_ARB	x y -	-	-
ARRAY1D	TEXTURE_1D_ARRAY_EXT	x - -	y	-
ARRAY2D	TEXTURE_2D_ARRAY_EXT	x y -	z	-
SHADOW1D	TEXTURE_1D	x - -	-	z
SHADOW2D	TEXTURE_2D	x y -	-	z
SHADOWRECT	TEXTURE_RECTANGLE_ARB	x y -	-	z
SHADOWCUBE	TEXTURE_CUBE_MAP	x y z	-	w
SHADOWARRAY1D	TEXTURE_1D_ARRAY_EXT	x - -	y	z
SHADOWARRAY2D	TEXTURE_2D_ARRAY_EXT	x y -	z	w
BUFFER	TEXTURE_BUFFER_EXT	<not supported>		

Table X.17: Texture types accessed for each of the <texTarget>, and coordinate mappings. The "SHADOW" and "ARRAY" targets are special pseudo-targets described below. The "coordinates used" column indicate the input values used for each coordinate of the texture lookup, the layer selector for array textures, and the reference value for texture comparisons. Buffer textures are not supported by normal texture lookup functions, but are supported by TXF and TXQ, described below.

Texture targets with "SHADOW" are used to access textures with a DEPTH_COMPONENT base internal format using depth comparisons (Section 3.8.14). Results of a texture access are undefined:

* if a "SHADOW" target is used, and the corresponding texture has a base internal format other than DEPTH_COMPONENT or a TEXTURE_COMPARE_MODE of NONE, or

* if a non-"SHADOW" target is used, and the corresponding texture has a base internal format of DEPTH_COMPONENT and a TEXTURE_COMPARE_MODE other than NONE.

If the texture being accessed is not complete (or cube complete for cubemap textures), no texture access is performed and the result is undefined.

A program will fail to load if it attempts to sample from multiple texture targets (including the SHADOW pseudo-targets) on the same texture image unit. For example, a program containing any two the following instructions will fail to load:

```
TEX out, coord, texture[0], 1D;
TEX out, coord, texture[0], 2D;
TEX out, coord, texture[0], ARRAY2D;
TEX out, coord, texture[0], SHADOW2D;
TEX out, coord, texture[0], 3D;
```

Additionally, multiple texture targets for a single texture image unit may not be used at the same time by the GL. The error INVALID_OPERATION is generated by Begin, RasterPos, or any command that performs an implicit Begin if an enabled program accesses one texture target for a texture unit while another enabled program or fixed-function fragment processing accesses a different texture target for the same texture image unit.

Some texture instructions use standard methods to compute partial derivatives and/or the level-of-detail used to perform texture accesses. For fragment programs, the functions

```
  float_vec ComputePartialsX(float_vec coord);
  float_vec ComputePartialsY(float_vec coord);
```

compute approximate component-wise partial derivatives of the floating-point vector <coord> relative to the X and Y coordinates, respectively. For vertex and geometry programs, these functions always return (0,0,0,0). The function

```
  float ComputeLOD(float_vec ddx, float_vec ddy);
```

maps partial derivative vectors <ddx> and <ddy> to ds/dx, dt/dx, dr/dx, ds/dy, dt/dy, and dr/dy and computes lambda_base(x,y) according to equation 3.18.

The TXF instruction provides the ability to extract a single texel from a specified texture image using the function

```
  result_t_vec TexelFetch(uint_vec coord, int_vec offset);
```

The extracted texel is converted to an (R,G,B,A) vector according to Table 3.21. The result vector is interpreted as floating-point, signed integer, or unsigned integer, according to the data type modifier of the instruction. If the internal format of the texture is not compatible with the instruction's data type modifer, the extracted texel value is undefined.

<coord> is a four-component signed integer vector used to identify the single texel accessed. The (i,j,k) coordinates of the texel and the layer used for array textures are extracted according to Table X.18. The level of detail accessed is obtained by adding the w component of <coord> to the base level (level_base). <offset> is a constant 3-component signed integer vector added to the texel coordinates prior to the texel fetch as described above. In addition to the restrictions described above, non-zero offset components are also not supported for BUFFER targets.

The texture used by TexelFetch() is specified by the image unit and target parameters provided in the instruction, as for TextureSample() above. Single texel fetches can not perform depth comparisons or access cubemaps. If a program contains a TXF instruction specifying one of the "SHADOW" or "CUBE" targets, it will fail to load.

texTarget	supported	coordinates used i j k	layer	lod
1D	yes	x - -	-	w
2D	yes	x y -	-	w
3D	yes	x y z	-	w
CUBE	no	- - -	-	-
RECT	yes	x y -	-	w
ARRAY1D	yes	x - -	y	w
ARRAY2D	yes	x y -	z	w
SHADOW1D	no	- - -	-	-
SHADOW2D	no	- - -	-	-
SHADOWRECT	no	- - -	-	-
SHADOWCUBE	no	- - -	-	-
SHADOWARRAY1D	no	- - -	-	-
SHADOWARRAY2D	no	- - -	-	-
BUFFER	yes	x - -	-	-

Table X.18, Mappings of texel fetch coordinates to texel location.

Single-texel fetches do not support LOD clamping or any texture wrap mode, and require a mipmapped minification filter to access any level of detail other than the base level. The results of the texel fetch are undefined:

* if the computed LOD is less than the texture's base level (level_base) or greater than the maximum level (level_max),

* if the computed LOD is not the texture's base level and the texture's minification filter is NEAREST or LINEAR,

* if the layer specified for array textures is negative or greater than the number of layers in the array texture,

* if the texel at (i,j,k) coordinates refer to a border texel outside the defined extents of the specified LOD, where

  ```
  i < -b_s, j < -b_s, k < -b_s,
  i >= w_s - b_s, j >= h_s - b_s, or k >= d_s - b_s,
  ```

  where the size parameters (w_s, h_s, d_s, and b_s) refer to the width, height, depth, and border size of the image, as in equations 3.15, 3.16, and 3.17, or

* if the texture being accessed is not complete (or cube complete for cubemaps).

Section 2.X.5, Program Flow Control

In addition to basic arithmetic, logical, and texture instructions, a number of flow control instructions are provided, which are described in detail in Section 2.X.8. Programs can contain several types of instruction blocks: IF/ELSE/ENDIF blocks, REP/ENDREP blocks, and subroutine blocks. IF/ELSE/ENDIF blocks are a set of instructions beginning with an "IF" instruction, ending with an "ENDIF" instruction, and possibly containing an optional "ELSE" instruction. REP/ENDREP blocks are a set of instructions beginning with a "REP" instruction and ending with an "ENDREP" instruction. Subroutine blocks begin with an instruction label identifying the name of the subroutine and ending just before the next instruction label or the end of the program. Examples include the following:

```
    MOVC CC, R0;
    IF GT.x;
      MOV R0, R1;      # executes if R0.x > 0
    ELSE;
      MOV R0, R2;      # executes if R0.x <= 0
    ENDIF;

    REP repCount;
    ADD R0, R0, R1;
    ENDREP;

  square:              # subroutine to compute R0^2
    MUL R0, R0, R0;
    RET;
  main:
    MOV R0, 9.0;
    CAL square;        # compute 9.0^2 in R0
```

IF/ELSE/ENDIF and REP/ENDREP blocks may be nested inside each other, and inside subroutines. In all cases, each instruction block must be terminated with the appropriate instruction (ENDIF for IF, ENDREP for REP). Nested instruction blocks must be wholly contained within a block -- if a REP instruction is found between an IF and ELSE instruction, the corresponding ENDREP must also be present between the IF and ELSE. Subroutines may not be nested inside IF/ELSE/ENDIF or REP/ENDREP blocks, or inside other subroutines. A program will fail to load if any instruction block is terminated by an incorrect instruction, is not terminated before the block containing it, or contains an instruction label.

IF/ELSE/ENDIF blocks evaluate a condition to determine which instructions to execute. If the condition is true, all instructions between the IF and ELSE are executed. If the condition is false, all instructions between the ELSE and ENDIF are executed. The ELSE instruction is optional. If the ELSE is omitted, all instructions between the IF and ENDIF are executed if the condition is true, or skipped if the condition is false. A limited amount of nesting is supported -- a program will fail to load if an IF instruction is nested inside MAX_PROGRAM_IF_DEPTH_NV or more IF/ELSE/ENDIF blocks.

REP/ENDREP blocks are used to execute a sequence of instructions multiple times. The REP instruction includes an optional scalar operand to specify a loop count indicating the number of times the block of instructions should be repeated. If the loop count is omitted, the contents of a REP/ENDREP block will be repeated indefinitely until the loop is explicitly terminated. A limited amount of nesting is supported -- a program will fail to load if a REP instruction is nested inside MAX_PROGRAM_LOOP_DEPTH_NV or more REP/ENDREP blocks.

Within a REP/ENDREP block, the CONT instruction can be used to terminate the current iteration of the loop by effectively jumping to the ENDREP instruction. The BRK instruction can be used to terminate the entire loop by effectively jumping to the instruction immediately following the ENDREP instruction. If CONT and BRK instructions are found inside multiply nested REP/ENDREP blocks, they apply to the innermost block. A program will fail to load if it includes a CONT or BRK instruction that is not contained inside a REP/ENDREP block.

A REP/ENDREP block without a specified loop count can result in an infinite loop. To prevent obvious infinite loops, a program will fail to load if it contains a REP/ENDREP block that contains neither a BRK instruction at the current nesting level or a RET instruction at any nesting level.

Subroutines are supported via the CAL and RET instructions. A subroutine block is identified by an instruction, which can be any valid identifier according to the <instLabel> grammar rule. The CAL instruction identifies a subroutine name to call according to the <instTarget> grammar rule. Instruction labels used in CAL instructions do not need to be defined in the program text that precedes the instruction, but a program will fail to load if it includes a CAL instruction that references an instruction label that is not defined anywhere in the program. When a CAL instruction is executed, it transfers control to the instruction immediately following the specified instruction label. Subsequent instructions in that subroutine are executed until a RET instruction is executed, or until program execution reaches another instruction label or the end of the program text. After the subroutine finishes, execution continues with the instruction immediately following the CAL instruction. When a RET instruction is issued, it will break out of any IF/ELSE/ENDIF or REP/ENDREP blocks that contain it.

Subroutines may call other subroutines before completing, up to an implementation-dependent maximum depth of MAX_PROGRAM_CALL_DEPTH_NV calls. Subroutines may call any subroutine in the program, including themselves, as long as the call depth limit is obeyed. The results of issuing a CAL instruction while MAX_PROGRAM_CALL_DEPTH subroutines have not completed has undefined results, including possible program termination.

Several flow control instructions include condition code tests. The IF instruction requires a condition test to determine what instructions are executed. The CONT, BRK, CAL, and RET instructions have an optional condition code test; if the test fails, the instructions are not executed. Condition code tests are specified by the <ccTest> grammar rule. The test is evaluated like the condition code write mask (section 2.X.4.3), and passes if and only if any of the four components passes.

If an instruction label named "main" is specified, GPU program execution begins with the instruction immediately following that label. Otherwise, it begins with the first instruction of the program. Instructions are executed in sequence until either a RET instruction is issued in the main subroutine or the end of the program text is reached.

Section 2.X.6, Program Options

Programs may specify a number of options to indicate that one or more extended language features are used by the program. All program options used by the program must be declared at the beginning of the program string. Each program option specified in a program string will modify the syntactic or semantic rules used to interpet the program and the execution environment used to execute the program. Features in program options not declared by the program are ignored, even if the option is otherwise supported by the GL. Each option declaration consists of two tokens: the keyword "OPTION" and an identifier.

The set of available options depends on the program type, and is enumerated in the specifications for each program type. Some program types may not provide any options.

Section 2.X.7, Program Declarations

Programs may include a number of declaration statements to specify characteristics of the program. Each declaration statement is followed by one or more arguments, separated by commas.

The set of available declarations depends on the program type, and is enumerated in the specifications for each program type. Some program types may not provide declarations.

Section 2.X.8, Program Instruction Set

The following sections enumerate the set of instructions supported for GPU programs.

Some instructions allow the use of one of the three basic data type modifiers (floating point, signed integer, and unsigned integer). Unless otherwise mentioned:

* the result and all of the operands will be interpreted according to the specified data type, and

* if no data type modifier is specified, the instruction will operate as though a floating-point modifier ("F") were specified.

Some instructions will override one or both of these rules.

Section 2.X.8.Z, ABS: Absolute Value

The ABS instruction performs a component-wise absolute value operation on the single operand to yield a result vector.

```
  tmp = VectorLoad(op0);
  result.x = abs(tmp.x);
  result.y = abs(tmp.y);
  result.z = abs(tmp.z);
  result.w = abs(tmp.w);
```

ABS supports all three data type modifiers. Taking the absolute value of an unsigned integer is not a useful operation, but is not illegal.

Section 2.X.8.Z, ADD: Add

The ADD instruction performs a component-wise add of the two operands to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = tmp0.x + tmp1.x;
  result.y = tmp0.y + tmp1.y;
  result.z = tmp0.z + tmp1.z;
  result.w = tmp0.w + tmp1.w;
```

ADD supports all three data type modifiers.

Section 2.X.8.Z, AND: Bitwise AND

The AND instruction performs a bitwise AND operation on the components of the two source vectors to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = tmp0.x & tmp1.x;
  result.y = tmp0.y & tmp1.y;
  result.z = tmp0.z & tmp1.z;
  result.w = tmp0.w & tmp1.w;
```

AND supports only signed and unsigned integer data type modifiers. If no type modifier is specified, both operands and the result are treated as signed integers.

Section 2.X.8.Z, BRK: Break out of Loop Instruction

The BRK instruction conditionally transfers control to the instruction immediately following the next ENDREP instruction. A BRK instruction has no effect if the condition code test evaluates to FALSE.

The following pseudocode describes the operation of the instruction:

```
  if (TestCC(cc.c***) || TestCC(cc.*c**) ||
      TestCC(cc.**c*) || TestCC(cc.***c)) {
    continue execution at instruction following the next ENDREP;
  }
```

Section 2.X.8.Z, CAL: Subroutine Call

The CAL instruction conditionally transfers control to the instruction following the label specified in the instruction. It also pushes a reference to the instruction immediately following the CAL instruction onto the call stack, where execution will continue after executing the matching RET instruction. The following pseudocode describes the operation of the instruction:

```
if (TestCC(cc.c***) || TestCC(cc.*c**) ||
    TestCC(cc.**c*) || TestCC(cc.***c)) {
  if (callStackDepth >= MAX_PROGRAM_CALL_DEPTH_NV) {
    // undefined results
  } else {
    callStack[callStackDepth] = nextInstruction;
    callStackDepth++;
  }
  // continue execution at instruction following <instTarget>
} else {
  // do nothing
}
```

In the pseudocode, <instTarget> is the label specified in the instruction matching the <branchLabel> grammar rule, <callStackDepth> is the current depth of the call stack, <callStack> is an array holding the call stack, and <nextInstruction> is a reference to the instruction immediately following the CAL instruction in the program string.

If the call stack overflows, the results of the CAL instruction are undefined, and can result in immediate program termination.

An instruction label signifies the beginning of a new subroutine. Subroutines may not nest or overlap. If a CAL instruction is executed and subsequent program execution reaches an instruction label before a corresponding RET instruction is executed, the subroutine call returns immediately, as though an unconditional RET instruction were inserted immediately before the instruction label.

(Note: On previous vertex program extensions -- NV_vertex_program2 and NV_vertex_program3 -- instruction labels were also used as targets for branch (BRA) instructions. This unstructured branching functionality has been replaced with the structured branching constructs found in this instruction set.)

Section 2.X.8.Z, CEIL: Ceiling

The CEIL instruction loads a single vector operand and performs a component-wise ceiling operation to generate a result vector.

```
tmp = VectorLoad(op0);
iresult.x = ceil(tmp.x);
iresult.y = ceil(tmp.y);
iresult.z = ceil(tmp.z);
iresult.w = ceil(tmp.w);
```

The ceiling operation returns the nearest integer greater than or equal to the operand. For example ceil(-1.7) = -1.0, ceil(+1.0) = +1.0, and ceil(+3.7) = +4.0.

CEIL supports all three data type modifiers. The single operand is always treated as a floating-point vector, but the result is written as a floating-point value, a signed integer, or an unsigned integer, as specified by the data type modifier. If a value is not exactly representable using the data type of the result (e.g., an overflow or writing a negative value to an unsigned integer), the result is undefined.

Section 2.X.8.Z, CMP: Compare

The CMP instructions performs a component-wise comparison of the first operand against zero, and copies the values of the second or third operands based on the results of the compare.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  tmp2 = VectorLoad(op2);
  result.x = (tmp0.x < 0) ? tmp1.x : tmp2.x;
  result.y = (tmp0.y < 0) ? tmp1.y : tmp2.y;
  result.z = (tmp0.z < 0) ? tmp1.z : tmp2.z;
  result.w = (tmp0.w < 0) ? tmp1.w : tmp2.w;
```

CMP supports all three data type modifiers. CMP with an unsigned data type modifier is not a useful operation, but is not illegal.

Section 2.X.8.Z, CONT: Continue with Next Loop Iteration

The CONT instruction conditionally transfers control to the next ENDREP instruction. A CONT instruction has no effect if the condition code test evaluates to FALSE.

The following pseudocode describes the operation of the instruction:

```
  if (TestCC(cc.c***) || TestCC(cc.*c**) ||
      TestCC(cc.**c*) || TestCC(cc.***c)) {
    continue execution at the next ENDREP;
  }
```

Section 2.X.8.Z, COS: Cosine with Reduction to [-PI,PI]

The COS instruction approximates the trigonometric cosine of the angle specified by the scalar operand and replicates it to all four components of the result vector. The angle is specified in radians and does not have to be in the range [-PI,PI].

```
  tmp = ScalarLoad(op0);
  result.x = ApproxCosine(tmp);
  result.y = ApproxCosine(tmp);
  result.z = ApproxCosine(tmp);
  result.w = ApproxCosine(tmp);
```

COS supports only floating-point data type modifiers.

Section 2.X.8.Z, DDX: Partial Derivative Relative to X

The DDX instruction computes approximate partial derivatives of a vector operand with respect to the X window coordinate, and is only available to fragment programs. See the NV_fragment_program4 specification for more details.

Section 2.X.8.Z, DDY: Partial Derivative Relative to Y

The DDY instruction computes approximate partial derivatives of a vector operand with respect to the Y window coordinate, and is only available to fragment programs. See the NV_fragment_program4 specification for more details.

Section 2.X.8.Z, DIV: Divide Vector Components by Scalar

The DIV instruction performs a component-wise divide of the first vector operand by the second scalar operand to produce a 4-component result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = ScalarLoad(op1);
  result.x = tmp0.x / tmp1;
  result.y = tmp0.y / tmp1;
  result.z = tmp0.z / tmp1;
  result.w = tmp0.w / tmp1;
```

DIV supports all three data type modifiers. For floating-point division, this instruction is not guaranteed to produce results identical to a RCP/MUL instruction sequence.

The results of an signed or unsigned integer division by zero are undefined.

Section 2.X.8.Z, DP2: 2-Component Dot Product

The DP2 instruction computes a two-component dot product of the two operands (using the first two components) and replicates the dot product to all four components of the result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  dot = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y);
  result.x = dot;
  result.y = dot;
  result.z = dot;
  result.w = dot;
```

DP2 supports only floating-point data type modifiers.

Section 2.X.8.Z, DP2A: 2-Component Dot Product with Scalar Add

The DP2 instruction computes a two-component dot product of the two operands (using the first two components), adds the x component of the third operand, and replicates the result to all four components of the result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  tmp2 = VectorLoad(op2);
  dot = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) + tmp2.x;
  result.x = dot;
  result.y = dot;
  result.z = dot;
  result.w = dot;
```

DP2A supports only floating-point data type modifiers.

Section 2.X.8.Z, DP3: 3-Component Dot Product

The DP3 instruction computes a three-component dot product of the two operands (using the x, y, and z components) and replicates the dot product to all four components of the result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  dot = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
        (tmp0.z * tmp1.z);
  result.x = dot;
  result.y = dot;
  result.z = dot;
  result.w = dot;
```

DP3 supports only floating-point data type modifiers.

Section 2.X.8.Z, DP4: 4-Component Dot Product

The DP4 instruction computes a four-component dot product of the two operands and replicates the dot product to all four components of the result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1):
  dot = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
        (tmp0.z * tmp1.z) + (tmp0.w * tmp1.w);
  result.x = dot;
  result.y = dot;
  result.z = dot;
  result.w = dot;
```

DP4 supports only floating-point data type modifiers.

Section 2.X.8.Z, DPH: Homogeneous Dot Product

The DPH instruction computes a three-component dot product of the two operands (using the x, y, and z components), adds the w component of the second operand, and replicates the sum to all four components of the result vector. This is equivalent to a four-component dot product where the w component of the first operand is forced to 1.0.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1):
  dot = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
        (tmp0.z * tmp1.z) + tmp1.w;
  result.x = dot;
  result.y = dot;
  result.z = dot;
  result.w = dot;
```

DPH supports only floating-point data type modifiers.

Section 2.X.8.Z, DST: Distance Vector

The DST instruction computes a distance vector from two specially-formatted operands. The first operand should be of the form [NA, d^2, d^2, NA] and the second operand should be of the form [NA, 1/d, NA, 1/d], where NA values are not relevant to the calculation and d is a vector length. If both vectors satisfy these conditions, the result vector will be of the form [1.0, d, d^2, 1/d].

The exact behavior is specified in the following pseudo-code:

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = 1.0;
  result.y = tmp0.y * tmp1.y;
  result.z = tmp0.z;
  result.w = tmp1.w;
```

Given an arbitrary vector, d^2 can be obtained using the DP3 instruction (using the same vector for both operands) and 1/d can be obtained from d^2 using the RSQ instruction.

This distance vector is useful for per-vertex light attenuation calculations: a DP3 operation using the distance vector and an attenuation constants vector as operands will yield the attenuation factor.

DST supports only floating-point data type modifiers.

Section 2.X.8.Z, ELSE: Start of If Test Else Block

The ELSE instruction signifies the end of the "execute if true" portion of an IF/ELSE/ENDIF block and the beginning of the "execute if false" portion.

If the condition evaluated at the IF statement was TRUE, when a program reaches the ELSE statement, it has completed the entire "execute if true" portion of the IF/ELSE/ENDIF block. Execution will continue at the corresponding ENDIF instruction.

If the condition evaluated at the IF statement was FALSE, program execution would skip over the entire "execute if true" portion of the IF/ELSE/ENDIF block, including the ELSE instruction.

Section 2.X.8.Z, EMIT: Emit Vertex

The EMIT instruction emits a new vertex to be added to the current output primitive generated by a geometry program, and is only available to geometry programs. See the NV_geometry_program4 specification for more details.

Section 2.X.8.Z, ENDIF: End of If Test Block

The ENDIF instruction signifies the end of an IF/ELSE/ENDIF block. It has no other effect on program execution.

Section 2.X.8,Z, ENDPRIM: End of Primitive

A geometry program can emit multiple primitives in a single invocation. The ENDPRIM instruction is used in a geometry program to signify the end of the current primitive and the beginning of a new primitive of the same type. It is only available to geometry programs. See the NV_geometry_program4 specification for more details.

Section 2.X.8.Z, ENDREP: End of Repeat Block

The ENDREP instruction specifies the end of a REP block.

When used with in conjunction with a REP instruction with a loop count, ENDREP decrements the loop counter. If the decremented loop counter is greater than zero, ENDREP transfers control to the instruction immediately after the corresponding REP instruction. If the loop counter is less than or equal to zero, execution continues at the instruction following the ENDREP instruction. When used in conjunction with a REP instruction without loop count, ENDREP always transfers control to the instruction immediately after the REP instruction.

```
  if (REP instruction includes a loop count) {
    LoopCount--;
    if (LoopCount > 0) {
      continue execution at instruction following corresponding REP
        instruction;
    }
  } else {
    continue execution at instruction following corresponding REP
      instruction;
  }
```

Section 2.X.8.Z, EX2: Exponential Base 2

The EX2 instruction approximates 2 raised to the power of the scalar operand and replicates the approximation to all four components of the result vector.

```
  tmp = ScalarLoad(op0);
  result.x = Approx2ToX(tmp);
  result.y = Approx2ToX(tmp);
  result.z = Approx2ToX(tmp);
  result.w = Approx2ToX(tmp);
```

EX2 supports only floating-point data type modifiers.

Section 2.X.8.Z, FLR: Floor

The FLR instruction loads a single vector operand and performs a component-wise floor operation to generate a result vector.

```
  tmp = VectorLoad(op0);
  result.x = floor(tmp.x);
  result.y = floor(tmp.y);
  result.z = floor(tmp.z);
  result.w = floor(tmp.w);
```

The floor operation returns the nearest integer less than or equal to the operand. For example floor(-1.7) = -2.0, floor(+1.0) = +1.0, and floor(+3.7) = +3.0.

FLR supports all three data type modifiers. The single operand is always treated as a floating-point value, but the result is written as a floating-point value, a signed integer, or an unsigned integer, as specified by the data type modifier. If a value is not exactly representable using the data type of the result (e.g., an overflow or writing a negative value to an unsigned integer), the result is undefined.

Section 2.X.8.Z, FRC: Fraction

The FRC instruction extracts the fractional portion of each component of the operand to generate a result vector. The fractional portion of a component is defined as the result after subtracting off the floor of the component (see FLR), and is always in the range [0.0, 1.0).

For negative values, the fractional portion is NOT the number written to the right of the decimal point -- the fractional portion of -1.7 is not 0.7 -- it is 0.3. 0.3 is produced by subtracting the floor of -1.7 (-2.0) from -1.7.

```
  tmp = VectorLoad(op0);
  result.x = fraction(tmp.x);
  result.y = fraction(tmp.y);
  result.z = fraction(tmp.z);
  result.w = fraction(tmp.w);
```

FRC supports only floating-point data type modifiers.

Section 2.X.8.Z, I2F: Integer to Float

The I2F instruction converts the components of an integer vector operand to floating-point to produce a floating-point result vector.

```
  tmp = VectorLoad(op0);
  result.x = (float) tmp.x;
  result.y = (float) tmp.y;
  result.z = (float) tmp.z;
  result.w = (float) tmp.w;
```

I2F supports only signed and unsigned integer data type modifiers. The single operand is interpreted according to the data type modifier. If no data type modifier is specified, the operand is treated as a signed integer vector. The result is always written as a float.

Section 2.X.8.Z, IF: Start of If Test Block

The IF instruction performs a condition code test to determine what instructions inside an IF/ELSE/ENDIF block are executed. If the test passes, execution continues at the instruction immediately following the IF instruction. If the test fails, IF transfers control to the instruction immediately following the corresponding ELSE instruction (if present) or the ENDIF instruction (if no ELSE is present).

Implementations may have a limited ability to nest IF blocks in any subroutine. If the number of IF/ENDIF blocks nested inside each other is MAX_PROGRAM_IF_DEPTH_NV or higher, a program will fail to compile.

```
  // Evaluate the condition.  If the condition is true, continue at the
  // next instruction.  Otherwise, continue at the
  if (TestCC(cc.c***) || TestCC(cc.*c**) ||
      TestCC(cc.**c*) || TestCC(cc.***c)) {
    continue execution at the next instruction;
  } else if (IF block contains an ELSE statement) {
    continue execution at instruction following corresponding ELSE;
  } else {
    continue execution at instruction following corresponding ENDIF;
  }
```

(Note: Unlike the NV_fragment_program2 extension, there is no run-time limit on the maximum overall depth of IF/ENDIF nesting. As long as each individual subroutine of the program obeys the static nesting limits, there will be no run-time errors in the program. With the NV_fragment_program2 extension, a program could terminate abnormally if it called a subroutine inside a very deeply nested set of IF/ENDIF blocks and the called subroutine also contained deeply nested IF/ENDIF blocks. SUch an error could occur even if neither subroutine exceeded static limits.)

Section 2.X.8.Z, KIL: Kill Fragment

The KIL instruction conditionally kills a fragment, and is only available to fragment programs. See the NV_fragment_program4 specification for more details.

Section 2.X.8.Z, LG2: Logarithm Base 2

The LG2 instruction approximates the base 2 logarithm of the scalar operand and replicates it to all four components of the result vector.

```
  tmp = ScalarLoad(op0);
  result.x = ApproxLog2(tmp);
  result.y = ApproxLog2(tmp);
  result.z = ApproxLog2(tmp);
  result.w = ApproxLog2(tmp);
```

If the scalar operand is zero or negative, the result is undefined.

LG2 supports only floating-point data type modifiers.

Section 2.X.8.Z, LIT: Compute Lighting Coefficients

The LIT instruction accelerates lighting computations by computing lighting coefficients for ambient, diffuse, and specular light contributions. The "x" component of the single operand is assumed to hold a diffuse dot product (n dot VP_pli, as in the vertex lighting equations in Section 2.13.1). The "y" component of the operand is assumed to hold a specular dot product (n dot h_i). The "w" component of the operand is assumed to hold the specular exponent of the material (s_rm), and is clamped to the range (-128, +128) exclusive.

The "x" component of the result vector receives the value that should be multiplied by the ambient light/material product (always 1.0). The "y" component of the result vector receives the value that should be multiplied by the diffuse light/material product (n dot VP_pli). The "z" component of the result vector receives the value that should be multiplied by the specular light/material product (f_i * (n dot h_i) ^ s_rm). The "w" component of the result is the constant 1.0.

Negative diffuse and specular dot products are clamped to 0.0, as is done in the standard per-vertex lighting operations. In addition, if the diffuse dot product is zero or negative, the specular coefficient is forced to zero.

```
  tmp = VectorLoad(op0);
  if (tmp.x < 0) tmp.x = 0;
  if (tmp.y < 0) tmp.y = 0;
  if (tmp.w < -(128.0-epsilon)) tmp.w = -(128.0-epsilon);
  else if (tmp.w > 128-epsilon) tmp.w = 128-epsilon;
  result.x = 1.0;
  result.y = tmp.x;
  result.z = (tmp.x > 0) ? RoughApproxPower(tmp.y, tmp.w) : 0.0;
  result.w = 1.0;
```

Since 0^0 is defined to be 1, RoughApproxPower(0.0, 0.0) will produce 1.0.

LIT supports only floating-point data type modifiers.

Section 2.X.8.Z, LRP: Linear Interpolation

The LRP instruction performs a component-wise linear interpolation between the second and third operands using the first operand as the blend factor.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  tmp2 = VectorLoad(op2);
  result.x = tmp0.x * tmp1.x + (1 - tmp0.x) * tmp2.x;
  result.y = tmp0.y * tmp1.y + (1 - tmp0.y) * tmp2.y;
  result.z = tmp0.z * tmp1.z + (1 - tmp0.z) * tmp2.z;
  result.w = tmp0.w * tmp1.w + (1 - tmp0.w) * tmp2.w;
```

LRP supports only floating-point data type modifiers.

Section 2.X.8.Z, MAD: Multiply and Add

The MAD instruction performs a component-wise multiply of the first two operands, and then does a component-wise add of the product to the third operand to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  tmp2 = VectorLoad(op2);
  result.x = tmp0.x * tmp1.x + tmp2.x;
  result.y = tmp0.y * tmp1.y + tmp2.y;
  result.z = tmp0.z * tmp1.z + tmp2.z;
  result.w = tmp0.w * tmp1.w + tmp2.w;
```

The multiplication and addition operations in this instruction are subject to the same rules as described for the MUL and ADD instructions.

MAD supports all three data type modifiers.

Section 2.X.8.Z, MAX: Maximum

The MAX instruction computes component-wise maximums of the values in the two operands to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x > tmp1.x) ? tmp0.x : tmp1.x;
  result.y = (tmp0.y > tmp1.y) ? tmp0.y : tmp1.y;
  result.z = (tmp0.z > tmp1.z) ? tmp0.z : tmp1.z;
  result.w = (tmp0.w > tmp1.w) ? tmp0.w : tmp1.w;
```

MAX supports all three data type modifiers.

Section 2.X.8.Z, MIN: Minimum

The MIN instruction computes component-wise minimums of the values in the two operands to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x > tmp1.x) ? tmp1.x : tmp0.x;
  result.y = (tmp0.y > tmp1.y) ? tmp1.y : tmp0.y;
  result.z = (tmp0.z > tmp1.z) ? tmp1.z : tmp0.z;
  result.w = (tmp0.w > tmp1.w) ? tmp1.w : tmp0.w;
```

MIN supports all three data type modifiers.

Section 2.X.8.Z, MOD: Modulus

The MOD instruction performs a component-wise modulus operation on the first vector operand by the second scalar operand to produce a 4-component result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = ScalarLoad(op1);
  result.x = tmp0.x % tmp1;
  result.y = tmp0.y % tmp1;
  result.z = tmp0.z % tmp1;
  result.w = tmp0.w % tmp1;
```

MOD supports both signed and unsigned integer data type modifiers. If no data type modifier is specified, both operands and the result are treated as signed integers.

Section 2.X.8.Z, MOV: Move

The MOV instruction copies the value of the operand to yield a result vector.

```
  result = VectorLoad(op0);
```

MOV supports all three data type modifiers.

Section 2.X.8.Z, MUL: Multiply

The MUL instruction performs a component-wise multiply of the two operands to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = tmp0.x * tmp1.x;
  result.y = tmp0.y * tmp1.y;
  result.z = tmp0.z * tmp1.z;
  result.w = tmp0.w * tmp1.w;
```

MUL supports all three data type modifiers. The MUL instruction additionally supports three special modifiers.

The "S24" and "U24" modifiers specify "fast" signed or unsigned integer multiplies of 24-bit quantities, respectively. The results of such multiplies are undefined if either operand is outside the range [-2^23,+2^23-1] for S24 or [0,2^24-1] for U24. If "S24" or "U24" is specified, the data type is implied and normal data type modifiers may not be provided.

The "HI" modifier specifies a 32-bit integer multiply that returns the 32 most significant bits of the 64-bit product. Integer multiplies without the "HI" modifier normally return the least significant bits of the product. If "HI" is specified, either of the "S" or "U" integer data type modifiers must also be specified.

Note that if condition code updates are performed on integer multiplies, the overflow or carry flags are always cleared, even if the product overflowed. If it is necessary to determine if the results of an integer multiply overflowed, the MUL.HI instruction may be used.

Section 2.X.8.Z, NOT: Bitwise Not

The NOT instruction performs a component-wise bitwise NOT operation on the source vector to produce a result vector.

```
  tmp = VectorLoad(op0);
  tmp.x = ~tmp.x;
  tmp.y = ~tmp.y;
  tmp.z = ~tmp.z;
  tmp.w = ~tmp.w;
```

NOT supports only integer data type modifiers. If no type modifier is specified, the operand and the result are treated as signed integers.

Section 2.X.8.Z, NRM: Normalize 3-Component Vector

The NRM instruction normalizes the vector given by the x, y, and z components of the vector operand to produce the x, y, and z components of the result vector. The w component of the result is undefined.

```
  tmp = VectorLoad(op0);
  scale = ApproxRSQ(tmp.x * tmp.x + tmp.y * tmp.y + tmp.z * tmp.z);
  result.x = tmp.x * scale;
  result.y = tmp.y * scale;
  result.z = tmp.z * scale;
  result.w = undefined;
```

NRM supports only floating-point data type modifiers.

Section 2.X.8.Z, OR: Bitwise Or

The OR instruction performs a bitwise OR operation on the components of the two source vectors to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = tmp0.x | tmp1.x;
  result.y = tmp0.y | tmp1.y;
  result.z = tmp0.z | tmp1.z;
  result.w = tmp0.w | tmp1.w;
```

OR supports only integer data type modifiers. If no type modifier is specified, both operands and the result are treated as signed integers.

Section 2.X.8.Z, PK2H: Pack Two 16-bit Floats

The PK2H instruction converts the "x" and "y" components of the single floating-point vector operand into 16-bit floating-point format, packs the bit representation of these two floats into a 32-bit unsigned integer, and replicates that value to all four components of the result vector. The PK2H instruction can be reversed by the UP2H instruction below.

```
  tmp0 = VectorLoad(op0);
  /* result obtained by combining raw bits of tmp0.x, tmp0.y */
  result.x = RawBits(tmp0.x) | (RawBits(tmp0.y) << 16);
  result.y = RawBits(tmp0.x) | (RawBits(tmp0.y) << 16);
  result.z = RawBits(tmp0.x) | (RawBits(tmp0.y) << 16);
  result.w = RawBits(tmp0.x) | (RawBits(tmp0.y) << 16);
```

PK2H supports all three data type modifiers. The single operand is always treated as a floating-point value, but the result is written as a floating-point value, a signed integer, or an unsigned integer, as specified by the data type modifier. For integer results, the bits can be interpreted as described above. For floating-point result variables, the packed results do not constitute a meaningful floating-point variable and should only be used to feed future unpack instructions.

A program will fail to load if it contains a PK2H instruction that writes its results to a variable declared as "SHORT".

Section 2.X.8.Z, PK2US: Pack Two Floats as Unsigned 16-bit

The PK2US instruction converts the "x" and "y" components of the single floating-point vector operand into a packed pair of 16-bit unsigned scalars. The scalars are represented in a bit pattern where all '0' bits corresponds to 0.0 and all '1' bits corresponds to 1.0. The bit representations of the two converted components are packed into a 32-bit unsigned integer, and that value is replicated to all four components of the result vector. The PK2US instruction can be reversed by the UP2US instruction below.

```
  tmp0 = VectorLoad(op0);
  if (tmp0.x < 0.0) tmp0.x = 0.0;
  if (tmp0.x > 1.0) tmp0.x = 1.0;
  if (tmp0.y < 0.0) tmp0.y = 0.0;
  if (tmp0.y > 1.0) tmp0.y = 1.0;
  us.x = round(65535.0 * tmp0.x);  /* us is a ushort vector */
  us.y = round(65535.0 * tmp0.y);
  /* result obtained by combining raw bits of us. */
  result.x = ((us.x) | (us.y << 16));
  result.y = ((us.x) | (us.y << 16));
  result.z = ((us.x) | (us.y << 16));
  result.w = ((us.x) | (us.y << 16));
```

PK2US supports all three data type modifiers. The single operand is always treated as a floating-point value, but the result is written as a floating-point value, a signed integer, or an unsigned integer, as specified by the data type modifier. For integer result variables, the bits can be interpreted as described above. For floating-point result variables, the packed results do not constitute a meaningful floating-point variable and should only be used to feed future unpack instructions.

A program will fail to load if it contains a PK2S instruction that writes its results to a variable declared as "SHORT".

Section 2.X.8.Z, PK4B: Pack Four Floats as Signed 8-bit

The PK4B instruction converts the four components of the single floating-point vector operand into 8-bit signed quantities. The signed quantities are represented in a bit pattern where all '0' bits corresponds to -128/127 and all '1' bits corresponds to +127/127. The bit representations of the four converted components are packed into a 32-bit unsigned integer, and that value is replicated to all four components of the result vector. The PK4B instruction can be reversed by the UP4B instruction below.

```
  tmp0 = VectorLoad(op0);
  if (tmp0.x < -128/127) tmp0.x = -128/127;
  if (tmp0.y < -128/127) tmp0.y = -128/127;
  if (tmp0.z < -128/127) tmp0.z = -128/127;
  if (tmp0.w < -128/127) tmp0.w = -128/127;
  if (tmp0.x > +127/127) tmp0.x = +127/127;
  if (tmp0.y > +127/127) tmp0.y = +127/127;
  if (tmp0.z > +127/127) tmp0.z = +127/127;
  if (tmp0.w > +127/127) tmp0.w = +127/127;
  ub.x = round(127.0 * tmp0.x + 128.0);  /* ub is a ubyte vector */
  ub.y = round(127.0 * tmp0.y + 128.0);
  ub.z = round(127.0 * tmp0.z + 128.0);
  ub.w = round(127.0 * tmp0.w + 128.0);
  /* result obtained by combining raw bits of ub. */
  result.x = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
  result.y = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
  result.z = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
  result.w = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
```

PK4B supports all three data type modifiers. The single operand is always treated as a floating-point value, but the result is written as a floating-point value, a signed integer, or an unsigned integer, as specified by the data type modifier. For integer result variables, the bits can be interpreted as described above. For floating-point result variables, the packed results do not constitute a meaningful floating-point variable and should only be used to feed future unpack instructions. A program will fail to load if it contains a PK4B instruction that writes its results to a variable declared as "SHORT".

Section 2.X.8.Z, PK4UB: Pack Four Floats as Unsigned 8-bit

The PK4UB instruction converts the four components of the single floating-point vector operand into a packed grouping of 8-bit unsigned scalars. The scalars are represented in a bit pattern where all '0' bits corresponds to 0.0 and all '1' bits corresponds to 1.0. The bit representations of the four converted components are packed into a 32-bit unsigned integer, and that value is replicated to all four components of the result vector. The PK4UB instruction can be reversed by the UP4UB instruction below.

```
  tmp0 = VectorLoad(op0);
  if (tmp0.x < 0.0) tmp0.x = 0.0;
  if (tmp0.x > 1.0) tmp0.x = 1.0;
  if (tmp0.y < 0.0) tmp0.y = 0.0;
  if (tmp0.y > 1.0) tmp0.y = 1.0;
  if (tmp0.z < 0.0) tmp0.z = 0.0;
  if (tmp0.z > 1.0) tmp0.z = 1.0;
  if (tmp0.w < 0.0) tmp0.w = 0.0;
  if (tmp0.w > 1.0) tmp0.w = 1.0;
  ub.x = round(255.0 * tmp0.x);  /* ub is a ubyte vector */
  ub.y = round(255.0 * tmp0.y);
  ub.z = round(255.0 * tmp0.z);
  ub.w = round(255.0 * tmp0.w);
  /* result obtained by combining raw bits of ub. */
  result.x = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
  result.y = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
  result.z = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
  result.w = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
```

PK4UB supports all three data type modifiers. The single operand is always treated as a floating-point value, but the result is written as a floating-point value, a signed integer, or an unsigned integer, as specified by the data type modifier. For integer result variables, the bits can be interpreted as described above. For floating-point result variables, the packed results do not constitute a meaningful floating-point variable and should only be used to feed future unpack instructions.

A program will fail to load if it contains a PK4UB instruction that writes its results to a variable declared as "SHORT".

Section 2.X.8.Z, POW: Exponentiate

The POW instruction approximates the value of the first scalar operand raised to the power of the second scalar operand and replicates it to all four components of the result vector.

```
  tmp0 = ScalarLoad(op0);
  tmp1 = ScalarLoad(op1);
  result.x = ApproxPower(tmp0, tmp1);
  result.y = ApproxPower(tmp0, tmp1);
  result.z = ApproxPower(tmp0, tmp1);
  result.w = ApproxPower(tmp0, tmp1);
```

The exponentiation approximation function may be implemented using the base 2 exponentiation and logarithm approximation operations in the EX2 and LG2 instructions. In particular,

```
  ApproxPower(a,b) = ApproxExp2(b * ApproxLog2(a)).
```

Note that a logarithm may be involved even for cases where the exponent is an integer. This means that it may not be possible to exponentiate correctly with a negative base. In constrast, it is possible in a "normal" mathematical formulation to raise negative numbers to integral powers (e.g., (-3)^2== 9, and (-0.5)^-2==4).

POW supports only floating-point data type modifiers.

Section 2.X.8.Z, RCC: Reciprocal (Clamped)

The RCC instruction approximates the reciprocal of the scalar operand, clamps the result to one of two ranges, and replicates the clamped result to all four components of the result vector.

If the approximated reciprocal is greater than 0.0, the result is clamped to the range [2^-64, 2^+64]. If the approximate reciprocal is not greater than zero, the result is clamped to the range [-2^+64, -2^-64].

```
  tmp = ScalarLoad(op0);
  result.x = ClampApproxReciprocal(tmp);
  result.y = ClampApproxReciprocal(tmp);
  result.z = ClampApproxReciprocal(tmp);
  result.w = ClampApproxReciprocal(tmp);
```

RCC supports only floating-point data type modifiers.

Section 2.X.8.Z, RCP: Reciprocal

The RCP instruction approximates the reciprocal of the scalar operand and replicates it to all four components of the result vector.

```
  tmp = ScalarLoad(op0);
  result.x = ApproxReciprocal(tmp);
  result.y = ApproxReciprocal(tmp);
  result.z = ApproxReciprocal(tmp);
  result.w = ApproxReciprocal(tmp);
```

RCP supports only floating-point data type modifiers.

Section 2.X.8.Z, REP: Start of Repeat Block

The REP instruction begins a REP/ENDREP block. The REP instruction supports an optional operand whose x component specifies the initial value for the loop count. The loop count indicates the number of times the instructions between the REP and corresponding ENDREP instruction will be executed. If the initial value of the loop count is not positive, the entire block is skipped and execution continues at the instruction following the corresponding ENDREP instruction. If the loop count is specified as a floating-point value, it is converted to the largest integer less than or equal to the specified value (i.e., taking its floor).

If no operand is provided to REP, the loop count is ignored and the corresponding ENDREP instruction unconditionally transfers control to the instruction immediately following the REP instruction. The only way to exit such a loop is with the BRK instruction. To prevent obvious infinite loops, a program that includes a REP/ENDREP block with no loop count will fail to compile unless it contains either a BRK instruction at the current nesting level or a RET instruction at any nesting level.

Implementations may have a limited ability to nest REP/ENDREP blocks. If the number of REP/ENDREP blocks nested inside each other is

MAX_PROGRAM_LOOP_DEPTH_NV or higher, a program will fail to compile.

```
  // Set up loop information for the new nesting level.
  tmp = VectorLoad(op0);
  LoopCount = floor(tmp.x);
  if (LoopCount <= 0) {
    continue execution at the corresponding ENDREP;
  }
```

REP supports all three data type modifiers. The single operand is interpreted according to the data type modifier.

(Note: Unlike the NV_fragment_program2 extension, REP blocks in this extension support fully general looping; the specified loop count can be computed in the program itself. Additionally, there is no run-time limit on the maximum overall depth of REP/ENDREP nesting. As long as each individual subroutine of the program obeys the static nesting limits, there will be no run-time errors in the program. With the NV_fragment_program2 extension, a program could terminate abnormally if it called a subroutine inside a deeply nested set of REP/ENDREP blocks and the called subroutine also contained deeply nested REP/ENDREP blocks. Such an error could occur even if neither subroutine exceeded static limits.)

Section 2.X.8.Z, RET: Subroutine Return

The RET instruction conditionally returns from a subroutine initiated by a CAL instruction by popping an instruction reference off the top of the call stack and transferring control to the referenced instruction. The following pseudocode describes the operation of the instruction:

```
  if (TestCC(cc.c***) || TestCC(cc.*c**) ||
      TestCC(cc.**c*) || TestCC(cc.***c)) {
    if (callStackDepth <= 0) {
      // terminate program
    } else {
      callStackDepth--;
      instruction = callStack[callStackDepth];
    }

    // continue execution at <instruction>
  } else {
    // do nothing
  }
```

In the pseudocode, <callStackDepth> is the depth of the call stack, <callStack> is an array holding the call stack, and <instruction> is a reference to an instruction previously pushed onto the call stack.

If the call stack is empty when RET executes, the program terminates normally.

Section 2.X.8.Z, RFL: Reflection Vector

The RFL instruction computes the reflection of the second vector operand (the "direction" vector) about the vector specified by the first vector operand (the "axis" vector). Both operands are treated as 3D vectors (the w components are ignored). The result vector is another 3D vector (the "reflected direction" vector). The length of the result vector, ignoring rounding errors, should equal that of the second operand.

```
  axis = VectorLoad(op0);
  direction = VectorLoad(op1);
  tmp.w = (axis.x * axis.x + axis.y * axis.y + axis.z * axis.z);
  tmp.x = (axis.x * direction.x + axis.y * direction.y +
           axis.z * direction.z);
  tmp.x = 2.0 * tmp.x;
  tmp.x = tmp.x / tmp.w;
  result.x = tmp.x * axis.x - direction.x;
  result.y = tmp.x * axis.y - direction.y;
  result.z = tmp.x * axis.z - direction.z;
```

RFL supports only floating-point data type modifiers.

Section 2.X.8.Z, ROUND: Round to Nearest Integer

The ROUND instruction loads a single vector operand and performs a component-wise round operation to generate a result vector.

```
  tmp = VectorLoad(op0);
  result.x = round(tmp.x);
  result.y = round(tmp.y);
  result.z = round(tmp.z);
  result.w = round(tmp.w);
```

The round operation returns the nearest integer to the operand. If the fractional portion of the operand is 0.5, round() selects the nearest even integer. For example round(-1.7) = -2.0, round(+1.0) = +1.0, and round(+3.7) = +4.0.

ROUND supports all three data type modifiers. The single operand is always treated as a floating-point value, but the result is written as a floating-point value, a signed integer, or an unsigned integer, as specified by the data type modifier. If a value is not exactly representable using the data type of the result (e.g., an overflow or writing a negative value to an unsigned integer), the result is undefined.

Section 2.X.8.Z, RSQ: Reciprocal Square Root

The RSQ instruction approximates the reciprocal of the square root of the scalar operand and replicates it to all four components of the result vector.

```
  tmp = ScalarLoad(op0);
  result.x = ApproxRSQRT(tmp);
  result.y = ApproxRSQRT(tmp);
  result.z = ApproxRSQRT(tmp);
  result.w = ApproxRSQRT(tmp);
```

If the operand is less than or equal to zero, the results of the instruction are undefined.

RSQ supports only floating-point data type modifiers.

Note that this instruction differs from the RSQ instruction in ARB_vertex_program in that it does not implicitly take the absolute value of its operand. The |abs| operator can be used to achieve equivalent semantics.

Section 2.X.8.Z, SAD: Sum of Absolute Differences

The SAD instruction performs a component-wise difference of the first two integer operands (subtracting the second from the first), and then does a component-wise add of the absolute value of the difference to the third unsigned integer operand to yield an unsigned integer result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  tmp2 = VectorLoad(op2);
  result.x = abs(tmp0.x - tmp1.x) + tmp2.x;
  result.y = abs(tmp0.y - tmp1.y) + tmp2.y;
  result.z = abs(tmp0.z - tmp1.z) + tmp2.z;
  result.w = abs(tmp0.w - tmp1.w) + tmp2.w;
```

SAD supports signed and unsigned integer data type modifiers. The first two operands are interpreted according to the data type modifier. The third operand and the result are always unsigned integers.

Section 2.X.8.Z, SCS: Sine/Cosine without Reduction

The SCS instruction approximates the trigonometric sine and cosine of the angle specified by the scalar operand and places the cosine in the x component and the sine in the y component of the result vector. The z and w components of the result vector are undefined. The angle is specified in radians and must be in the range [-PI,PI].

```
  tmp = ScalarLoad(op0);
  result.x = ApproxCosine(tmp);
  result.y = ApproxSine(tmp);
```

If the scalar operand is not in the range [-PI,PI], the result vector is undefined.

SCS supports only floating-point data type modifiers.

Section 2.X.8.Z, SEQ: Set on Equal

The SEQ instruction performs a component-wise comparison of the two operands. Each component of the result vector returns a TRUE value (described below) if the corresponding component of the first operand is equal to that of the second, and a FALSE value otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x == tmp1.x) ? TRUE : FALSE;
  result.y = (tmp0.y == tmp1.y) ? TRUE : FALSE;
  result.z = (tmp0.z == tmp1.z) ? TRUE : FALSE;
  result.w = (tmp0.w == tmp1.w) ? TRUE : FALSE;
```

SEQ supports all data type modifiers. For floating-point data types, the TRUE value is 1.0 and the FALSE value is 0.0. For signed integer data types, the TRUE value is -1 and the FALSE value is 0. For unsigned integer data types, the TRUE value is the maximum integer value (all bits are ones) and the FALSE value is zero.

Section 2.X.8.Z, SFL: Set on False

The SFL instruction is a degenerate case of the other "Set on" instructions that sets all components of the result vector to a FALSE value (described below).

```
  result.x = FALSE;
  result.y = FALSE;
  result.z = FALSE;
  result.w = FALSE;
```

SFL supports all data type modifiers. For floating-point data types, the FALSE value is 0.0. For signed and unsigned integer data types, the FALSE value is zero.

Section 2.X.8.Z, SGE: Set on Greater Than or Equal

The SGE instruction performs a component-wise comparison of the two operands. Each component of the result vector returns a TRUE value (described below) if the corresponding component of the first operand is greater than or equal to that of the second, and a FALSE value otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x >= tmp1.x) ? TRUE : FALSE;
  result.y = (tmp0.y >= tmp1.y) ? TRUE : FALSE;
  result.z = (tmp0.z >= tmp1.z) ? TRUE : FALSE;
  result.w = (tmp0.w >= tmp1.w) ? TRUE : FALSE;
```

SGE supports all data type modifiers. For floating-point data types, the TRUE value is 1.0 and the FALSE value is 0.0. For signed integer data types, the TRUE value is -1 and the FALSE value is 0. For unsigned integer data types, the TRUE value is the maximum integer value (all bits are ones) and the FALSE value is zero.

Section 2.X.8.Z, SGT: Set on Greater Than

The SGT instruction performs a component-wise comparison of the two operands. Each component of the result vector returns a TRUE value (described below) if the corresponding component of the first operand is greater than that of the second, and a FALSE value otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x > tmp1.x) ? TRUE : FALSE;
  result.y = (tmp0.y > tmp1.y) ? TRUE : FALSE;
  result.z = (tmp0.z > tmp1.z) ? TRUE : FALSE;
  result.w = (tmp0.w > tmp1.w) ? TRUE : FALSE;
```

SGT supports all data type modifiers. For floating-point data types, the TRUE value is 1.0 and the FALSE value is 0.0. For signed integer data types, the TRUE value is -1 and the FALSE value is 0. For unsigned integer data types, the TRUE value is the maximum integer value (all bits are ones) and the FALSE value is zero.

Section 2.X.8.Z, SHL: Shift Left

The SHL instruction performs a component-wise left shift of the bits of the first operand by the value of the second scalar operand to produce a result vector. The bits vacated during the shift operation are filled with zeroes.

```
  tmp0 = VectorLoad(op0);
  tmp1 = ScalarLoad(op1);
  result.x = tmp0.x << tmp1;
  result.y = tmp0.y << tmp1;
  result.z = tmp0.z << tmp1;
  result.w = tmp0.w << tmp1;
```

The results of a shift operation ("<<") are undefined if the value of the second operand is negative, or greater than or equal to the number of bits in the first operand.

SHL supports both signed and unsigned integer data type modifiers. If no modifier is provided, the operands and the result are treated as signed integers.

Section 2.X.8.Z, SHR: Shift Right

The SHR instruction performs a component-wise right shift of the bits of the first operand by the value of the second scalar operand to produce a result vector. The bits vacated during shift operation are filled with zeros if the operand is non-negative and ones otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = ScalarLoad(op1);
  result.x = tmp0.x >> tmp1;
  result.y = tmp0.y >> tmp1;
  result.z = tmp0.z >> tmp1;
  result.w = tmp0.w >> tmp1;
```

The results of a shift operation (">>") are undefined if the value of the second operand is negative, or greater than or equal to the number of bits in the first operand.

SHR supports both signed and unsigned integer data type modifiers. If no modifiers are provided, the operands and the result are treated as signed integers.

Section 2.X.8.Z, SIN: Sine with Reduction to [-PI,PI]

The SIN instruction approximates the trigonometric sine of the angle specified by the scalar operand and replicates it to all four components of the result vector. The angle is specified in radians and does not have to be in the range [-PI,PI].

```
  tmp = ScalarLoad(op0);
  result.x = ApproxSine(tmp);
  result.y = ApproxSine(tmp);
  result.z = ApproxSine(tmp);
  result.w = ApproxSine(tmp);
```

SIN supports only floating-point data type modifiers.

Section 2.X.8.Z, SLE: Set on Less Than or Equal

The SLE instruction performs a component-wise comparison of the two operands. Each component of the result vector returns a TRUE value (described below) if the corresponding component of the first operand is less than or equal to that of the second, and a FALSE value otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x <= tmp1.x) ? TRUE : FALSE;
  result.y = (tmp0.y <= tmp1.y) ? TRUE : FALSE;
  result.z = (tmp0.z <= tmp1.z) ? TRUE : FALSE;
  result.w = (tmp0.w <= tmp1.w) ? TRUE : FALSE;
```

SLE supports all data type modifiers. For floating-point data types, the TRUE value is 1.0 and the FALSE value is 0.0. For signed integer data types, the TRUE value is -1 and the FALSE value is 0. For unsigned integer data types, the TRUE value is the maximum integer value (all bits are ones) and the FALSE value is zero.

Section 2.X.8.Z, SLT: Set on Less Than

The SLT instruction performs a component-wise comparison of the two operands. Each component of the result vector returns a TRUE value (described below) if the corresponding component of the first operand is less than that of the second, and a FALSE value otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x < tmp1.x) ? TRUE : FALSE;
  result.y = (tmp0.y < tmp1.y) ? TRUE : FALSE;
  result.z = (tmp0.z < tmp1.z) ? TRUE : FALSE;
  result.w = (tmp0.w < tmp1.w) ? TRUE : FALSE;
```

SLT supports all data type modifiers. For floating-point data types, the TRUE value is 1.0 and the FALSE value is 0.0. For signed integer data types, the TRUE value is -1 and the FALSE value is 0. For unsigned integer data types, the TRUE value is the maximum integer value (all bits are ones) and the FALSE value is zero.

Section 2.X.8.Z, SNE: Set on Not Equal

The SNE instruction performs a component-wise comparison of the two operands. Each component of the result vector returns a TRUE value (described below) if the corresponding component of the first operand is less than that of the second, and a FALSE value otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x != tmp1.x) ? TRUE : FALSE;
  result.y = (tmp0.y != tmp1.y) ? TRUE : FALSE;
  result.z = (tmp0.z != tmp1.z) ? TRUE : FALSE;
  result.w = (tmp0.w != tmp1.w) ? TRUE : FALSE;
```

SNE supports all data type modifiers. For floating-point data types, the TRUE value is 1.0 and the FALSE value is 0.0. For signed integer data types, the TRUE value is -1 and the FALSE value is 0. For unsigned integer data types, the TRUE value is the maximum integer value (all bits are ones) and the FALSE value is zero.

Section 2.X.8.Z, SSG: Set Sign

The SSG instruction generates a result vector containing the signs of each component of the single vector operand. Each component of the result vector is 1.0 if the corresponding component of the operand is greater than zero, 0.0 if the corresponding component of the operand is equal to zero, and -1.0 if the corresponding component of the operand is less than zero.

```
  tmp = VectorLoad(op0);
  result.x = SetSign(tmp.x);
  result.y = SetSign(tmp.y);
  result.z = SetSign(tmp.z);
  result.w = SetSign(tmp.w);
```

SSG supports only floating-point data type modifiers.

Section 2.X.8.Z, STR: Set on True

The STR instruction is a degenerate case of the other "Set on" instructions that sets all components of the result vector to a TRUE value (described below).

```
  result.x = TRUE;
  result.y = TRUE;
  result.z = TRUE;
  result.w = TRUE;
```

STR supports all data type modifiers. For floating-point data types, the TRUE value is 1.0. For signed integer data types, the TRUE value is -1. For unsigned integer data types, the TRUE value is the maximum integer value (all bits are ones).

Section 2.X.8.Z, SUB: Subtract

The SUB instruction performs a component-wise subtraction of the second operand from the first to yield a result vector.

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1);
result.x = tmp0.x - tmp1.x;
result.y = tmp0.y - tmp1.y;
result.z = tmp0.z - tmp1.z;
result.w = tmp0.w - tmp1.w;
```

SUB supports all three data type modifiers.

Section 2.X.8.Z, SWZ: Extended Swizzle

The SWZ instruction loads the single vector operand, and performs a swizzle operation more powerful than that provided for loading normal vector operands to yield an instruction vector.

After the operand is loaded, the "x", "y", "z", and "w" components of the result vector are selected by the first, second, third, and fourth matches of the <extSwizComp> pattern in the <extendedSwizzle> rule.

A result component can be selected from any of the four components of the operand or the constants 0.0 and 1.0. The result component can also be optionally negated. The following pseudocode describes the component selection method. "operand" refers to the vector operand, "select" is an enumerant where the values ZERO, ONE, X, Y, Z, and W correspond to the <extSwizSel> rule matching "0", "1", "x", "y", "z", and "w", respectively. "negate" is TRUE if and only if the <optionalSign> rule in <extSwizComp> matches "-".

```
float ExtSwizComponent(floatVec operand, enum select, boolean negate)
{
    float result;
    switch (select) {
      case ZERO:  result = 0.0; break;
      case ONE:   result = 1.0; break;
      case X:     result = operand.x; break;
      case Y:     result = operand.y; break;
      case Z:     result = operand.z; break;
      case W:     result = operand.w; break;
    }
    if (negate) {
      result = -result;
    }
    return result;
}
```

The entire extended swizzle operation is then defined using the following pseudocode:

```
  tmp = VectorLoad(op0);
  result.x = ExtSwizComponent(tmp, xSelect, xNegate);
  result.y = ExtSwizComponent(tmp, ySelect, yNegate);
  result.z = ExtSwizComponent(tmp, zSelect, zNegate);
  result.w = ExtSwizComponent(tmp, wSelect, wNegate);
```

"xSelect", "xNegate", "ySelect", "yNegate", "zSelect", "zNegate", "wSelect", and "wNegate" correspond to the "select" and "negate" values above for the four <extSwizComp> matches.

Since this instruction allows for component selection and negation for each individual component, the grammar does not allow the use of the normal swizzle and negation operations allowed for vector operands in other instructions.

SWZ supports only floating-point data type modifiers.

Section 2.X.8.Z, TEX: Texture Sample

The TEX instruction takes the four components of a single floating-point source vector and performs a filtered texture access as described in Section 2.X.4.4. The returned (R,G,B,A) value is written to the floating-point result vector. Partial derivatives and the level of detail are computed automatically.

```
  tmp = VectorLoad(op0);
  ddx = ComputePartialsX(tmp);
  ddy = ComputePartialsY(tmp);
  lambda = ComputeLOD(ddx, ddy);
  result = TextureSample(tmp, lambda, ddx, ddy, texelOffset);
```

TEX supports all three data type modifiers. The single operand is always treated as a floating-point vector; the results are interpreted according to the data type modifier.

Section 2.X.8.Z, TRUNC: Truncate (Round Toward Zero)

The TRUNC instruction loads a single vector operand and performs a component-wise truncate operation to generate a result vector.

```
  tmp = VectorLoad(op0);
  result.x = trunc(tmp.x);
  result.y = trunc(tmp.y);
  result.z = trunc(tmp.z);
  result.w = trunc(tmp.w);
```

The truncate operation returns the nearest integer to zero smaller in magnitude than the operand. For example trunc(-1.7) = -1.0, trunc(+1.0) = +1.0, and trunc(+3.7) = +3.0.

TRUNC supports all three data type modifiers. The single operand is always treated as a floating-point value, but the result is written as a floating-point value, a signed integer, or an unsigned integer, as specified by the data type modifier. If a value is not exactly representable using the data type of the result (e.g., an overflow or writing a negative value to an unsigned integer), the result is undefined.

Section 2.X.8.Z, TXB: Texture Sample with Bias

The TXB instruction takes the four components of a single floating-point source vector and performs a filtered texture access as described in Section 2.X.4.4. The returned (R,G,B,A) value is written to the floating-point result vector. Partial derivatives and the level of detail are computed automatically, but the fourth component of the source vector is added to the computed LOD prior to sampling.

```
  tmp = VectorLoad(op0);
  ddx = ComputePartialsX(tmp);
  ddy = ComputePartialsY(tmp);
  lambda = ComputeLOD(ddx, ddy);
  result = TextureSample(tmp, lambda + tmp.w, ddx, ddy, texelOffset);
```

The single source vector in the TXB instruction does not have enough coordinates to specify a lookup into a two-dimensional array texture or cube map texture with both an LOD bias and an explicit reference value for depth comparison. A program will fail to load if it contains a TXB instruction with a target of SHADOWCUBE or SHADOWARRAY2D.

TXB supports all three data type modifiers. The single operand is always treated as a floating-point vector; the results are interpreted according to the data type modifier.

Section 2.X.8.Z, TXD: Texture Sample with Partials

The TXD instruction takes the four components of the first floating-point source vector and performs a filtered texture access as described in Section 2.X.4.4. The returned (R,G,B,A) value is written to the floating-point result vector. The partial derivatives of the texture coordinates with respect to X and Y are specified by the second and third floating-point source vectors. The level of detail is computed automatically using the provided partial derivatives.

Note that for cube map texture targets, the provided partial derivatives are in the coordinate system used before texture coordinates are projected onto the appropriate cube face. The partial derivatives of the post-projection texture coordinates, which are used for level-of-detail and anisotropic filtering calculations, are derived from the original coordinates and partial derivatives in an implementation-dependent manner.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  tmp2 = VectorLoad(op2);
  lambda = ComputeLOD(tmp1, tmp2);
  result = TextureSample(tmp0, lambda, tmp1, tmp2, texelOffset);
```

TXD supports all three data type modifiers. All three operands are always treated as floating-point vectors; the results are interpreted according to the data type modifier.

Section 2.X.8.Z, TXF: Texel Fetch

The TXF instruction takes the four components of a single signed integer source vector and performs a single texel fetch as described in Section 2.X.4.4. The first three components provide the <i>, <j>, and <k> values for the texel fetch, and the fourth component is used to determine the LOD to access. The returned (R,G,B,A) value is written to the floating-point result vector. Partial derivatives are irrelevant for single texel fetches.

```
  tmp = VectorLoad(op0);
  result = TexelFetch(tmp, texelOffset);
```

TXF supports all three data type modifiers. The single vector operand is treated as a signed integer vector; the results are interpreted according to the data type modifier.

Section 2.X.8.Z, TXL: Texture Sample with LOD

The TXL instruction takes the four components of a single floating-point source vector and performs a filtered texture access as described in Section 2.X.4.4. The returned (R,G,B,A) value is written to the floating-point result vector. The level of detail is taken from the fourth component of the source vector.

Partial derivatives are not computed by the TXL instruction and anisotropic filtering is not performed.

```
  tmp = VectorLoad(op0);
  ddx = (0,0,0);
  ddy = (0,0,0);
  result = TextureSample(tmp, tmp.w, ddx, ddy, texelOffset);
```

The single source vector in the TXL instruction does not have enough coordinates to specify a lookup into a 2D array or cube map texture with both an explicit LOD and a reference value for depth comparison. A program will fail to load if it contains a TXL instruction with a target of SHADOWCUBE or SHADOWARRAY2D.

TXL supports all three data type modifiers. The single vector operand is treated as a floating-point vector; the results are interpreted according to the data type modifier.

Section 2.X.8.Z, TXP: Texture Sample with Projection

The TXP instruction divides the first three components of its single floating-point source vector by its fourth component, maps the results to s, t, and r, and performs a filtered texture access as described in Section 2.X.4.4. The returned (R,G,B,A) value is written to the floating-point result vector. Partial derivatives and the level of detail are computed automatically.

```
  tmp0 = VectorLoad(op0);
  tmp0.x = tmp0.x / tmp0.w;
  tmp0.y = tmp0.y / tmp0.w;
  tmp0.z = tmp0.z / tmp0.w;
  ddx = ComputePartialsX(tmp);
  ddy = ComputePartialsY(tmp);
  lambda = ComputeLOD(ddx, ddy);
  result = TextureSample(tmp, lambda, ddx, ddy, texelOffset);
```

The single source vector in the TXP instruction does not have enough coordinates to specify a lookup into a 2D array or cube map texture with both a Q coordinate and an explicit reference value for depth comparison. A program will fail to load if it contains a TXP instruction with a target of SHADOWCUBE or SHADOWARRAY2D.

TXP supports all three data type modifiers. The single vector operand is treated as a floating-point vector; the results are interpreted according to the data type modifier.

Section 2.X.8.Z, TXQ: Texture Size Query

The TXQ instruction takes the first component of the single integer vector operand, adds the number of the base level of the specified texture to determine a texture image level, and returns an integer result vector containing the size of the image at that level of the texture.

For one-dimensional and one-dimensional array textures, the "x" component of the result vector is filled with the width of the image(s). For two-dimensional, rectangle, cube map, and two-dimensional array textures, the "x" and "y" components are filled with the width and height of the image(s). For three-dimensional textures, the "x", "y", and "z" components are filled with the width, height, and depth of the image. Additionally, the number of layers in an array texture is returned in the "y" component of the result for one-dimensional array textures or the "z" component for two-dimensional array textures. All other components of the result vector is undefined. For the purposes of this instruction, the width, height, and depth of a texture do NOT include any border.

```
  tmp0 = VectorLoad(op0);
  tmp0.x = tmp0.x + texture[op1].target[op2].base_level;
  result.x = texture[op1].target[op2].level[tmp0.x].width;
  result.y = texture[op1].target[op2].level[tmp0.x].height;
  result.z = texture[op1].target[op2].level[tmp0.x].depth;
```

If the level computed by adding the operand to the base level of the texture is less than the base level number or greater than the maximum level number, the results are undefined.

TXQ supports no data type modifiers; the scalar operand and the result vector are both interpreted as signed integers.

Section 2.X.8.Z, UP2H: Unpack Two 16-bit Floats

The UP2H instruction unpacks two 16-bit floats stored together in a 32-bit scalar operand. The first 16-bit float (stored in the 16 least significant bits) is written into the "x" and "z" components of the result vector; the second is written into the "y" and "w" components of the result vector.

This operation undoes the type conversion and packing performed by the PK2H instruction.

```
  tmp = ScalarLoad(op0);
  result.x = (fp16) (RawBits(tmp) & 0xFFFF);
  result.y = (fp16) ((RawBits(tmp) >> 16) & 0xFFFF);
  result.z = (fp16) (RawBits(tmp) & 0xFFFF);
  result.w = (fp16) ((RawBits(tmp) >> 16) & 0xFFFF);
```

UP2H supports all three data type modifiers. The single operand is read as a floating-point value, a signed integer, or an unsigned integer, as specified by the data type modifier; the 32 least significant bits of the encoding are used for unpacking. For floating-point operand variables, it is expected (but not required) that the operand was produced by a previous pack instruction. The result is always written as a floating-point vector.

A program will fail to load if it contains a UP2H instruction whose operand is a variable declared as "SHORT".

Section 2.X.8.Z, UP2US: Unpack Two Unsigned 16-bit Integers

The UP2US instruction unpacks two 16-bit unsigned values packed together in a 32-bit scalar operand. The unsigned quantities are encoded where a bit pattern of all '0' bits corresponds to 0.0 and a pattern of all '1' bits corresponds to 1.0. The "x" and "z" components of the result vector are obtained from the 16 least significant bits of the operand; the "y" and "w" components are obtained from the 16 most significant bits.

This operation undoes the type conversion and packing performed by the PK2US instruction.

```
  tmp = ScalarLoad(op0);
  result.x = ((RawBits(tmp) >> 0)  & 0xFFFF) / 65535.0;
  result.y = ((RawBits(tmp) >> 16) & 0xFFFF) / 65535.0;
  result.z = ((RawBits(tmp) >> 0)  & 0xFFFF) / 65535.0;
  result.w = ((RawBits(tmp) >> 16) & 0xFFFF) / 65535.0;
```

UP2US supports all three data type modifiers. The single operand is read as a floating-point value, a signed integer, or an unsigned integer, as specified by the data type modifier; the 32 least significant bits of the encoding are used for unpacking. For floating-point operand variables, it is expected (but not required) that the operand was produced by a previous pack instruction. The result is always written as a floating-point vector.

A GPU program will fail to load if it contains a UP2S instruction
whose operand is a variable declared as "SHORT".

Section 2.X.8.Z, UP4B: Unpack Four Signed 8-bit Integers

The UP4B instruction unpacks four 8-bit signed values packed together
in a 32-bit scalar operand. The signed quantities are encoded where
a bit pattern of all '0' bits corresponds to -128/127 and a pattern
of all '1' bits corresponds to +127/127. The "x" component of the
result vector is the converted value corresponding to the 8 least
significant bits of the operand; the "w" component corresponds to
the 8 most significant bits.

This operation undoes the type conversion and packing performed by
the PK4B instruction.

```
  tmp = ScalarLoad(op0);
  result.x = (((RawBits(tmp) >> 0) & 0xFF) - 128) / 127.0;
  result.y = (((RawBits(tmp) >> 8) & 0xFF) - 128) / 127.0;
  result.z = (((RawBits(tmp) >> 16) & 0xFF) - 128) / 127.0;
  result.w = (((RawBits(tmp) >> 24) & 0xFF) - 128) / 127.0;
```

UP2B supports all three data type modifiers. The single operand is read
as a floating-point value, a signed integer, or an unsigned integer, as
specified by the data type modifier; the 32 least significant bits of the
encoding are used for unpacking. For floating-point operand variables, it
is expected (but not required) that the operand was produced by a previous
pack instruction. The result is always written as a floating-point
vector.

A program will fail to load if it contains a UP4B instruction whose
operand is a variable declared as "SHORT".

Section 2.X.8.Z, UP4UB: Unpack Four Unsigned 8-bit Integers

The UP4UB instruction unpacks four 8-bit unsigned values packed
together in a 32-bit scalar operand. The unsigned quantities are
encoded where a bit pattern of all '0' bits corresponds to 0.0 and a
pattern of all '1' bits corresponds to 1.0. The "x" component of the
result vector is obtained from the 8 least significant bits of the
operand; the "w" component is obtained from the 8 most significant
bits.

This operation undoes the type conversion and packing performed by
the PK4UB instruction.

```
  tmp = ScalarLoad(op0);
  result.x = ((RawBits(tmp) >> 0)  & 0xFF) / 255.0;
  result.y = ((RawBits(tmp) >> 8)  & 0xFF) / 255.0;
  result.z = ((RawBits(tmp) >> 16) & 0xFF) / 255.0;
  result.w = ((RawBits(tmp) >> 24) & 0xFF) / 255.0;
```

UP4UB supports all three data type modifiers. The single operand is read
as a floating-point value, a signed integer, or an unsigned integer, as
specified by the data type modifier; the 32 least significant bits of the
encoding are used for unpacking. For floating-point operand variables, it

is expected (but not required) that the operand was produced by a previous pack instruction. The result is always written as a floating-point vector.

A program will fail to load if it contains a UP4UB instruction whose operand is a variable declared as "SHORT".

Section 2.X.8.Z, X2D: 2D Coordinate Transformation

The X2D instruction multiplies the 2D offset vector specified by the "x" and "y" components of the second vector operand by the 2x2 matrix specified by the four components of the third vector operand, and adds the transformed offset vector to the 2D vector specified by the "x" and "y" components of the first vector operand. The first component of the sum is written to the "x" and "z" components of the result; the second component is written to the "y" and "w" components of the result.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  tmp2 = VectorLoad(op2);
  result.x = tmp0.x + tmp1.x * tmp2.x + tmp1.y * tmp2.y;
  result.y = tmp0.y + tmp1.x * tmp2.z + tmp1.y * tmp2.w;
  result.z = tmp0.x + tmp1.x * tmp2.x + tmp1.y * tmp2.y;
  result.w = tmp0.y + tmp1.x * tmp2.z + tmp1.y * tmp2.w;
```

X2D supports only floating-point data type modifiers.

Section 2.X.8.Z, XOR: Exclusive Or

The XOR instruction performs a bitwise XOR operation on the components of the two source vectors to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = tmp0.x ^ tmp1.x;
  result.y = tmp0.y ^ tmp1.y;
  result.z = tmp0.z ^ tmp1.z;
  result.w = tmp0.w ^ tmp1.w;
```

XOR supports only integer data type modifiers. If no type modifier is specified, both operands and the result are treated as signed integers.

Section 2.X.8.Z, XPD: Cross Product

The XPD instruction computes the cross product using the first three components of its two vector operands to generate the x, y, and z components of the result vector. The w component of the result vector is undefined.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = tmp0.y * tmp1.z - tmp0.z * tmp1.y;
  result.y = tmp0.z * tmp1.x - tmp0.x * tmp1.z;
  result.z = tmp0.x * tmp1.y - tmp0.y * tmp1.x;
```

XPD supports only floating-point data type modifiers.

Additions to Chapter 3 of the OpenGL 1.5 Specification (Rasterization)

Modify Section 3.8.1, Texture Image Specification, p. 150

(modify 4th paragraph, p. 151 -- add cubemaps to the list of texture targets that can be used with DEPTH_COMPONENT textures) Textures with a base internal format of DEPTH_COMPONENT are supported by texture image specification commands only if <target> is TEXTURE_1D, TEXTURE_2D, TEXTURE_CUBE_MAP, TEXTURE_RECTANGLE_ARB, TEXTURE_1D_ARRAY_EXT, TEXTURE_2D_ARRAY_EXT, PROXY_TEXTURE_1D PROXY_TEXTURE_2D, PROXY_TEXTURE_CUBE_MAP, PROXY_TEXTURE_RECTANGLE_ARB, PROXY_TEXTURE_1D_ARRAY_EXT, or PROXY_TEXTURE_2D_ARRAY_EXT. Using this format in conjunction with any other target will result in an INVALID_OPERATION error.

Delete Section 3.8.7, Texture Wrap Modes. (The language in this section is folded into updates to the following section, and is no longer needed here.)

Modify Section 3.8.8, Texture Minification:

(replace the last paragraph, p. 171): Let s(x,y) be the function that associates an s texture coordinate with each set of window coordinates (x,y) that lie within a primitive; define t(x,y) and r(x,y) analogously. Let

```
  u(x,y) = w_t * s(x,y) + offsetu_shader,
  v(x,y) = h_t * t(x,y) + offsetv_shader,
  w(x,y) = d_t * r(x,y) + offsetw_shader, and
```

where w_t, h_t, and d_t are as defined by equations 3.15, 3.16, and 3.17 with w_s, h_s, and d_s equal to the width, height, and depth of the image array whose level is level_base. (offsetu_shader, offsetv_shader, offsetw_shader) is the texel offset specified in the vertex, geometry, or fragment program instruction used to perform the access. For fixed-function texture accesses, all three shader offsets are taken to be zero. For a one-dimensional texture, define v(x,y) == 0 and w(x,y) === 0; for two-dimensional textures, define w(x,y) == 0.

(start a new paragraph with "For a polygon, rho is given at a fragment with window coordinates...", and then continue with the original spec text.)

(replace text starting with the last paragraph on p. 172, continuing to the end of p. 174)

The (u,v,w) coordinates are then modified according the texture wrap modes, as specified in Table X.19, to generate a new set of coordinates (u',v',w').

TEXTURE_WRAP_S	Coordinate Transformation
CLAMP	u' = clamp(u, 0, w_t-0.5), if NEAREST filtering, clamp(u, 0, w_t), otherwise
CLAMP_TO_EDGE	u' = clamp(u, 0.5, w_t-0.5)
CLAMP_TO_BORDER	u' = clamp(u, -0.5, w_t+0.5)
REPEAT	u' = clamp(fmod(u, w_t), 0.5, w_t-0.5)
MIRROR_CLAMP_EXT	u' = clamp(fabs(u), 0.5, w_t-0.5), if NEAREST filtering, or = clamp(fabs(u), 0.5, w_t), otherwise
MIRROR_CLAMP_TO_EDGE_EXT	u' = clamp(fabs(u), 0.5, w_t-0.5)
MIRROR_CLAMP_TO_BORDER_EXT	u' = clamp(fabs(u), 0.5, w_t+0.5)
MIRRORED_REPEAT	u' = w_t - clamp(fabs(w_t - fmod(u, 2*w_t)), 0.5, w_t-0.5),

Table X.19: Texel coordinate wrap mode application. clamp(a,b,c) returns b if a<b, c if a>c, and a otherwise. fmod(a,b) returns a-b*floor(a/b), and fabs(a) returns the absolute value of a. For the v and w coordinates, TEXTURE_WRAP_T and h_t, and TEXTURE_WRAP_R and d_t, respectively, are used.

When lambda indicates minification, the value assigned to TEXTURE_MIN_FILTER is used to determine how the texture value for a fragment is selected.

When TEXTURE_MIN_FILTER is NEAREST, the texel in the image array of level level_base that is nearest (in Manhattan distance) to that specified by (s,t,r) is obtained. For a three-dimensional texture, the texel at location (i,j,k) becomes the texture value. For a two-dimensional texture, k is irrelevant, and the texel at location (i,j) becomes the texture value. For a one-dimensional texture, j and k are irrelevant, and the texel at location i becomes the texture value.

If the selected (i,j,k), (i,j), or i location refers to a border texel that satisfies any of the following conditions:

```
  i < -b_s,
  j < -b_s,
  k < -b_s,
  i >= w_l + b_s,
  j >= h_l + b_s, or
  j >= d_l + b_s,
```

then the border values defined by TEXTURE_BORDER_COLOR are used in place of the non-existent texel. If the texture contains color components, the values of TEXTURE_BORDER_COLOR are interpreted as an RGBA color to match the texture’s internal format in a manner consistent with table 3.15. If the texture contains depth components, the first component of TEXTURE_BORDER_COLOR is interpreted as a depth value.

When TEXTURE_MIN_FILTER is LINEAR, a 2x2x2 cube of texels in the image array of level level_base is selected. Let:

```
  i_0   = floor(u' - 0.5),
  j_0   = floor(v' - 0.5),
  k_0   = floor(w' - 0.5),
  i_1   = i_0 + 1,
  j_1   = j_0 + 1,
  k_1   = k_0 + 1,
  alpha = frac(u' - 0.5),
  beta  = frac(v' - 0.5),
  gamma = frac(w' - 0.5),
```

For a three-dimensional texture, the texture value tau is found as...

(replace last paragraph, p.174) For any texel in the equation above that refers to a border texel outside the defined range of the image, the texel value is taken from the texture border color as with NEAREST filtering.

Modify Section 3.8.14, Texture Comparison Modes (p. 185)

(modify 2nd paragraph, p. 188, indicating that the Q texture coordinate is used for depth comparisons on cubemap textures)

Let D_t be the depth texture value, in the range [0, 1]. For fixed-function texture lookups, let R be the interpolated <r> texture coordinate, clamped to the range [0, 1]. For texture lookups generated by a program instruction, let R be the reference value for depth comparisons provided in the instruction, also clamped to [0, 1]. Then the effective texture value L_t, I_t, or A_t is computed as follows:

Additions to Chapter 4 of the OpenGL 1.5 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 1.5 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 1.5 Specification (State and State Requests)

Modify Section 6.1.12 of the ARB_vertex_program specification.

(Add new integer program parameter queries, plus language that program environment or local parameter query results are undefined if the query specifies a data type incompatible with the data type of the parameter being queried.)

The commands

```
  void GetProgramEnvParameterdvARB(enum target, uint index,
                                   double *params);
  void GetProgramEnvParameterfvARB(enum target, uint index,
                                   float *params);
  void GetProgramEnvParameterIivNV(enum target, uint index,
                                   int *params);
  void GetProgramEnvParameterIuivNV(enum target, uint index,
                                    uint *params);
```

obtain the current value for the program environment parameter numbered <index> for the given program target <target>, and places the information in the array <params>. The values returned are undefined if the data type of the components of the parameter is not compatible with the data type of <params>. Floating-point components are compatible with "double" or "float"; signed and unsigned integer components are compatible with "int" and "uint", respectively. The error INVALID_ENUM is generated if <target> specifies a nonexistent program target or a program target that does not support program environment parameters. The error INVALID_VALUE is generated if <index> is greater than or equal to the implementation-dependent number of supported program environment parameters for the program target.

...

The commands

```
      void GetProgramLocalParameterdvARB(enum target, uint index,
                                    double *params);
      void GetProgramLocalParameterfvARB(enum target, uint index,
                                    float *params);
      void GetProgramLocalParameterIivNV(enum target, uint index,
                                    int *params);
      void GetProgramLocalParameterIuivNV(enum target, uint index,
                                     uint *params);
```

obtain the current value for the program local parameter numbered <index> belonging to the program object currently bound to <target>, and places the information in the array <params>. The values returned are undefined if the data type of the components of the parameter is not compatible with the data type of <params>. Floating-point components are compatible with "double' or "float"; signed and unsigned integer components are compatible with "int" and "uint", respectively. The error INVALID_ENUM is generated if <target> specifies a nonexistent program target or a program target that does not support program local parameters. The error INVALID_VALUE is generated if <index> is greater than or equal to the implementation-dependent number of supported program local parameters for the program target.

...

The command

```
      void GetProgramivARB(enum target, enum pname, int *params);
```

obtains program state for the program target <target>, writing ...

(add new paragraphs describing the new supported queries)

If <pname> is PROGRAM_ATTRIB_COMPONENTS_NV or PROGRAM_RESULT_COMPONENTS_NV, GetProgramivARB returns a single integer holding the number of active attribute or result variable components, respectively, used by the program object currently bound to <target>.

If <pname> is MAX_PROGRAM_ATTRIB_COMPONENTS or MAX_PROGRAM_RESULT_COMPONENTS_NV, GetProgramivARB returns a single integer holding the maximum number of active attribute or result variable components, respectively, supported for programs of type <target>.

Additions to Appendix A of the OpenGL 1.5 Specification (Invariance)

None.

Additions to the AGL/GLX/WGL Specifications

None.

GLX Protocol

None.

Errors

The error INVALID_VALUE is generated by ProgramLocalParameter4fARB, ProgramLocalParameter4fvARB, ProgramLocalParameter4dARB, ProgramLocalParameter4dvARB, ProgramLocalParameterI4iNV, ProgramLocalParameterI4ivNV, ProgramLocalParameterI4uiNV, ProgramLocalParameterI4uivNV, GetProgramLocalParameter4fvARB, GetProgramLocalParameter4dvARB, GetProgramLocalParameterI4ivNV, and GetProgramLocalParameterI4uivNV if <index> is greater than or equal to the number of program local parameters supported by <target>.

The error INVALID_VALUE is generated by ProgramEnvParameter4fARB, ProgramEnvParameter4fvARB, ProgramEnvParameter4dARB, ProgramEnvParameter4dvARB, ProgramEnvParameterI4iNV, ProgramEnvParameterI4ivNV, ProgramEnvParameterI4uiNV, ProgramEnvParameterI4uivNV, GetProgramEnvParameter4fvARB, GetProgramEnvParameter4dvARB, GetProgramEnvParameterI4ivNV, and GetProgramEnvParameterI4uivNV if <index> is greater than or equal to the number of program environment parameters supported by <target>.

The error INVALID_VALUE is generated by ProgramLocalParameters4fvNV, ProgramLocalParametersI4ivNV, and ProgramLocalParametersI4uivNV if the sum of <index> and <count> is greater than the number of program local parameters supported by <target>.

The error INVALID_VALUE is generated by ProgramEnvParameters4fvNV, ProgramEnvParametersI4ivNV, and ProgramEnvParametersI4uivNV if the sum of <index> and <count> is greater than the number of program environment parameters supported by <target>.

Dependencies on NV_parameter_buffer_object

If NV_parameter_buffer_object is not supported, references to program parameter buffer variables and bindings should be removed.

Dependencies on ARB_texture_rectangle

If ARB_texture_rectangle is not supported, references to rectangle textures and the RECT and SHADOWRECT texture target identifiers should be removed.

Dependencies on EXT_gpu_program_parameters

If EXT_gpu_program_parameters is not supported, references to the Program{Local,Env}Parameters4fvNV commands, which set multiple program local or environment parameters in a single call, should be removed. These prototypes were included in this spec for completeness only.

Dependencies on EXT_texture_integer

If EXT_texture_integer is not supported, references to texture lookups returning integer values in Section 2.X.4.4 (Texture Access) should be removed, and all texture formats are considered to produce floating-point values.

Dependencies on EXT_texture_array

If EXT_texture_array is not supported, references to array textures in Section 2.X.4.4 (Texture Access) and elsewhere should be removed, as should all references to the "ARRAY1D", "ARRAY2D", "SHADOWARRAY1D", and "SHADOWARRAY2D" tokens.

Dependencies on EXT_texture_buffer_object

If EXT_texture_buffer_object is not supported, references to buffer textures in Section 2.X.4.4 (Texture Access) and elsewhere should be removed, as should all references to the "BUFFER" tokens.

Dependencies on NV_primitive_restart

If NV_primitive_restart is supported, index values causing a primitive restart are not considered as specifying an End command, followed by another Begin. Primitive restart is therefore not guaranteed to immediately update bindings for material properties changed inside a Begin/End. The spec language says they "are not guaranteed to update program parameter bindings until the following End command."

New State

Get Value	Type	Get Command	Initial Value	Description	Sec	Attrib
PROGRAM_ATTRIB_COMPONENTS_NV	Z+	GetProgramivARB	-	number of components used for attributes	6.1.12	-
PROGRAM_RESULT_COMPONENTS_NV	Z+	GetProgramivARB	-	number of components used for results	6.1.12	-

Table X.20. New Program Object State. Program object queries return attributes of the program object currently bound to the program target <target>.

New Implementation Dependent State

Get Value	Type	Get Command	Minimum Value	Description	Sec.	Attrib
MIN_PROGRAM_TEXEL_OFFSET_EXT	Z	GetIntegerv	-8	minimum texel offset allowed in lookup	2.x.4.4	-
MAX_PROGRAM_TEXEL_OFFSET_EXT	Z	GetIntegerv	+7	maximum texel offset allowed in lookup	2.x.4.4	-
MAX_PROGRAM_ATTRIB_COMPONENTS_NV	Z+	GetProgramivARB	(*)	maximum number of components allowed for attributes	6.1.12	-
MAX_PROGRAM_RESULT_COMPONENTS_NV	Z+	GetProgramivARB	(*)	maximum number of components allowed for results	6.1.12	-
MAX_PROGRAM_GENERIC_ATTRIBS_NV	Z+	GetProgramivARB	(*)	number of generic attribute vectors supported	6.1.12	-
MAX_PROGRAM_GENERIC_RESULTS_NV	Z+	GetProgramivARB	(*)	number of generic result vectors supported	6.1.12	-
MAX_PROGRAM_CALL_DEPTH_NV	Z+	GetProgramivARB	4	maximum program call stack depth	2.X.5	-
MAX_PROGRAM_IF_DEPTH_NV	Z+	GetProgramivARB	48	maximum program if nesting	2.X.5	-
MAX_PROGRAM_LOOP_DEPTH_NV	Z+	GetProgramivARB	4	maximum program loop nesting	2.X.5	-

Table X.21: New Implementation-Dependent Values Introduced by NV_gpu_program4. (*) means that the required minimum is program type-specific. There are separate limits for each program type.

Issues

(1) How does this extension differ from previous NV_vertex_program and NV_fragment_program extensions?

RESOLVED:

- This extension provides a uniform set of instructions and bindings. Unlike previous extensions, the set of instructions and bindings available is generally the same. The only exceptions are a small number of instructions and bindings that make sense for one specific program type.

- This extension supports integer data types and provides a full-fledged integer instruction set.

- This extension supports array variables of all types, including temporaries. Array variables can be accessed directly or indirectly (using integer temporaries as indices).

- This extension provides a uniform set of structured branching constructs (if tests, loops, subroutines) that fully support run-time condition testing. Previous versions of NV_vertex_program provided unstructured branching. Previous versions of NV_fragment_program provided structure branching constructs, but the support was more limited -- for example, looping constructs couldn't specify loop counts with values computed at run time.

- This extension supports geometry programs, which are described in more detail in the NV_geometry_program4 extension.

- This extension provides the ability to specify and use cubemap textures with a DEPTH_COMPONENT internal format. Shadow mapping is supported; the Q texture coordinate is used as the reference value for comparisons.

(2) Is this extension backward-compatible with previous NV_vertex_program and NV_fragment_program extensions? If not, what support has been removed?

RESOLVED: This extension is largely, but not completely, backward-compatible. Functionality removed includes:

- Unstructured branching: NV_vertex_program2 included a general branch instruction "BRA" that could be used to jump to an arbitrary instruction. The "CAL" instruction could "call" to an arbitrary instruction into code that was not necessarily structured as simple subroutine blocks. Arbitrary unstructured branching can be difficult to implement efficiently on highly parallel GPU architectures, while basic structured branching is not nearly as difficult.

  This extension retains the "CAL" instruction but treats each block of code between instruction labels as a separate subroutine. The "BRA" instruction and arbitrary branching has been removed. The structured branching constructs in this extension are sufficient to implement almost all of the looping/branching support in high-level languages ("goto" being the most obvious exception).

- Address registers: NV_vertex_program added the notion of address registers, which were effectively under-powered integer temporaries. The set of instructions used to manipulate address registers was severely limited. NV_vertex_program[23] extended the original scalars to vectors and added a few more instructions to manipulate address registers. Fragment programs had no address registers until NV_fragment_program2 added the loop counter, which was very similar in functionality to vertex program address registers, but even more limited. This extension adds true integer temporaries, which can accomplish everything old address registers could do, and much more. Address register support was removed to simplify the API.

- NV_fragment_program2 LOOP construct: NV_fragment_program2 added a LOOP instruction, which let you repeat a block of code <N> times, with a parallel loop counter that started at <A> and stepped by <B> on each iteration. This construct was signficantly limited in several ways -- the loop count had to be constant, and you could only access the innermost loop counter in a nested loop. This extension discards the support and retains the simpler "REP" construct to implement loops. If desired, a loop counter can be implemented by manipulating an integer temporary. The "BRK" instruction (conditional break) is retained, and a "CONT" instruction (conditional continue) is added. Additionally, the loop count need not be a constant.

- NV_vertex_program and ARB_vertex_program EXP and LOG instructions: NV_vertex_program provided EXP and LOG instructions that computed a rough approximation of 2^x or log_2(x) and provided some additional values that could help refine the approximation. Those opcodes were carried forward into ARB_vertex_program. Both ARB_vertex_program and NV_vertex_program2 provided EX2 and LG2 instructions that computed a better approximation. All fragment program extensions also provided EX2 and LG2, but did not bother to include EXP and LOG. On the hardware targeted by this extension, there is no advantage to using EXP and LOG, so these opcodes have been removed for simplicity.

- NV_vertex_program3 and NV_fragment_program2 provide the ability to do indirect addressing of inputs/outputs when using bindings in instructions -- for example:

  ```
  MOV R0, vertex.attrib[A0.x+2];      # vertex
  MOV result.texcoord[A0.y], R1;      # vertex
  MOV R2, fragment.texcoord[A0.x];    # fragment
  ```

  This extension provides indexing capability, but using named array variables instead.

  ```
  ATTRIB attribs[] = { vertex.attrib[2..5] };
  MOV R0, attribs[A0.x];
  OUTPUT outcoords[] = { result.texcoord[0..3] };
  MOV outcoords[A0.y], R1;
  ATTRIB texcoords[] = { fragment.texcoord[0..2] };
  MOV R2, texcoords[A0.x];
  ```

  This approach makes the set of attribute and result bindings more regular. Additionally, it helps the assembler determine which vertex/fragment attributes are actually needed -- when the assembler sees constructs like "fragment.texcoord[A0.x]", it must treat *all* texture coordinates as live unless it can determine the range of values used for indexing. The named array variable approach explicitly identifies which attributes are needed when indexing is used.

Functionality altered includes:

- The RSQ instruction in the original NV_vertex_program and ARB_vertex_program extensions implicitly took the absolute value of their operand. Since the ARB extensions don't have numerics guarantees, computing the reciprocal square root of a negative value was not meaningful. To allow for the possibility of taking the reciprocal square root of a negative value (which should yield NaN -- "not a number"), the RSQ instruction in this instruction no longer implicitly takes the absolute value of its operand. Equivalent functionality can be achieved using the explicit |abs| absolute value operator on the operand to RSQ.

- The results of texture lookups accessing inconsistent textures are now undefined, instead of producing a fixed constant vector.

(3) What should this set of extensions be called?

RESOLVED: NV_gpu_program4, NV_vertex_program4, NV_fragment_program4, and NV_geometry_program4. Only NV_gpu_program4 will appear in the extension string; the other three specifications exist simply to define vertex, fragment, and geometry program-specific features.

The "gpu_program" name was chosen due to the common instruction set intended to run on GPUs. On previous chip generations, the vertex and fragment instruction sets were similar, but there were enough differences to package them separately.

The choice of "4" indicates that this is the fourth generation of programmable hardware from NVIDIA. The GeForce3 and GeForce4 series supported NV_vertex_program. The GeForce FX series supported NV_vertex_program2 and added fragment programmability with NV_fragment_program. Around this time, the OpenGL Architecture Review Board (ARB) approved ARB_vertex_program and ARB_fragment_program extensions, and NVIDIA added NV_vertex_program2_option and NV_fragment_program_option extensions exposing GeForce FX features using the ARB extensions' instruction set. The GeForce6 and GeForce7 series brought the NV_vertex_program3 and NV_fragment_program2 extensions, which extend the ARB extensions further. This extension adds geometry programs, and brings the "version number" for each of these extensions up to "4".

(4) This instruction adds integer data type support in programmable shaders that were previously float-centric. Should applications be able to pass integer values directly to the shaders, and if so, how does it work?

RESOLVED: The diagram at the bottom of this issue depicts data flows in the GL, as extended by this and related extensions.

This extension generalizes some state to be "typeless", instead of being strongly typed (and almost invariably floating-point) as in the core specification. We introduce a new set of functions to specify GL state as signed or unsigned integer values, instead of floating point values. These functions include:

* VertexAttribI*{i,ui}() -- Specify generic vertex attributes as integers. This extension does not create "integer" versions for fixed-function attribute functions (e.g., glColor, glTexCoord), which remain fully floating-point.

* Program{Env,Local}ParameterI*{i,ui}() -- Specify environment and local parameters as integers.

* TexImage*() with EXT_texture_integer internal formats -- Specify texture images as containing integer data whose values are not converted to floating-point values.

* EXT_parameter_buffer_object functions -- Bind (typeless) buffer object data stores for use as program parameters. These buffer objects can be loaded with either integer or floating-point data.

* EXT_texture_buffer_object functions -- Bind (typeless) buffer object data stores for use as textures. These buffer objects can be loaded with either integer or floating-point data.

Each type of program (using NV_gpu_program4 and related extension) can read attributes using any data type (float, signed integer, unsigned integer) and write result values used by subsequent stages using any data type.

Finally, there are several new places where integer data can be consumed by the GL:

* NV_transform_feedback -- Stream transformed vertex attribute components to a (typeless) buffer object. The transformed attributes can be written as signed or unsigned integers in vertex and geometry programs.

* EXT_texture_integer internal formats and framebuffer objects -- Provide support for rendering to integer texture formats, where final fragment values are treated as signed or unsigned integers, rather than floating-point values.

The diagram below represents a substantial portion of the GL pipeline. Each line connecting blocks represents an interface where data is "produced" from the GL state or by fixed-function or programmable pipeline stages and "consumed" by another pipeline stage. Each producer and consumer is labeled with a data type. For producers, the "(typeless)" designation generally means that the state and/or output can be written as floating-point values or as signed or unsigned integers. "(float)" means that the outputs are always written as floating-point. The same distinction applies to consumers -- "(typeless)" means that the consumer is capable of reading inputs using any data type, and "(float)" means that consumer always reads inputs as floating-point values.

To get sane results, applications must ensure that each value passed between pipeline stages is produced and consumed using the same data type. If a value is written in one stage as a floating-point value; it must be read as a floating-point value as well. If such a value is read as a signed or unsigned integer, its value is considered undefined. In practice, the raw bits used to represent the floating-point (IEEE single-precision floating-point encoding in the initial implementation of this spec) will be treated as an integer.

Type matching between stages is not enforced by the GL, because the overhead of doing so would be substantial. Such overhead would include:

* matching the inputs and outputs of each pipeline stage (fixed-function or programmable) every time the program configuration or fixed-function state changes,

* tracking the data type of each generic vertex attribute and checking it against the vertex program's inputs,

* tracking the data type of each program parameter and checking it against the manner the parameters were used in programs,

* matching color buffers against fragment program outputs.

Such error checking is certainly valuable, but the additional CPU overhead cost is substantial. Given that current CPUs often have a hard time keeping up with high-end GPUs, adding more overhead is a step in the wrong direction. We expect developer tools, such as instrumented drivers, to be able to provide type checking on most interfaces.

The diagram below depicts assembly programmability. Using vertex, geometry, and fragment shaders provided by the OpenGL Shading Language (GLSL) isn't substantially different from the assembly interface, except that the interfaces between programmable pipeline stages are more tightly coupled in GLSL (vertex, geometry, and fragment shaders are linked together into a single program object), and that shader variables are more strongly typed in GLSL than in the assembly interface.

In the figure below, the first programmable stage is vertex program execution. For all inputs read by the vertex program, they must be specified in the GL vertex APIs (immediate mode or vertex arrays) using a data type matching the data type read by the shader. Additionally, vertex programs (and all other program types) can read program parameters, parameter buffers, and textures. In all cases the parameter, buffer, or texture data must be accessed in the shader using the same data type used to specify the data. If vertex programs are disabled, fixed-function vertex processing is used. Fixed-function vertex processing is fully floating-point, and all the conventional vertex attributes and state used by fixed-function are floating-point values.

After vertex processing, an optional geometry program can be executed, which reads attributes written by vertex programs (or fixed-functon) and writes out new vertex attributes. The vertex attributes it reads must have been written by the vertex program (or fixed-function) using a matching data type.

After geometry program execution, vertex attributes can optionally be written out to buffer objects using the NV_transform_feedback extension. The vertex attributes are written by the GL to the buffer objects using the same data type used to write the attribute in the geometry program (or vertex program if geometry programs are disabled).

Then, rasterization generates fragments based on transformed vertices. Most attributes written by vertex or geometry programs can be read by fragment programs, after the rasterization hardware "interpolates" them. This extension allows fragment programs to control how each attribute is interpolated. If an attribute is flat-shaded, it will be taken from the output attribute of the provoking vertex of the primitive using the same data type. If an attribute is smooth-shaded, the per-vertex attributes will be interpreted as a floating-point value, and a floating-point result. One necessary consequence of this is that any integer per-fragment attributes must be flat-shaded. To prevent some interpolation type errors, assembly and GLSL fragment shaders will not compile if they declare an integer fragment attribute that is not flat shaded. [NOTE: While point primitives generally have constant attributes, any integer attributes must still be flat-shaded; point rasterization may perform (degenerate) floating-point interpolation.]

Fragment programs must read attributes using data types matching the outputs of the interpolation or flat-shading operations. They may write one or more color outputs using any data type, but the data type used must match the corresponding framebuffer attachments. Outputs directed at signed or unsigned integer textures (EXT_texture_integer) must be written using the appropriate integer data type; all other outputs must be written as floating-point values. Note that some of the fixed-function per-fragment operations (e.g., blending, alpha test) are specified as floating-point operations and are skipped when directed at signed or unsigned integer color buffers.

```
                                 generic               conventional
                                 vertex                  vertex
                                attributes              attributes
                                   | (typeless)             | (float)
                                   |                        |
                                   |                        |
                                   | +----------------------+
     program                       | |                      |
    parameters ----+               | |                      |
    (typeless)     |               | | (typeless)           | (float)
                   |               V V                      V
     constant      +-+----------> vertex              fixed-function
     buffers   ----+ |(typeless)  program                 vertex
    (typeless)     | |              |                       |
                   | |              | (typeless)            | (float)
     textures  ----+ |              V                       |
    (typeless)       |              |<----------------------+
        |            |              |
        |            |              +---------------+
        |            |              |               |
        |            |              | (typeless)    |
        |            |              V               |
        |            +---------> geometry           |
        |            |(typeless) program            |
        |            |              |               |
        |            |              | (typeless)    |
        |            |              V               |
        |            |              |<--------------+
        |            |              |
        |            |              |
        |            |              +-----------------+
        |            |              |                 |(typeless)
        |            |              |                 v
        |            |              |             transform
        |            |              |             feedback
        |            |              |              buffers
        |            |              |
        |            |              |
        |            |              +-----------------------+
        |            |              |                       |
        |            |              | (float)               | (typeless)
        |            |              V                       V
        |            |         interpolated               flat
        |            |          attributes             attributes
        |            |              |                       |
        |            |              | (float)               | (typeless)
        |            |              V                       |
        |            |              |<----------------------+
        |            |              |
        |            |              +-----------------------+
        |            |              |                       |
        |            |              | (typeless)            | (float)
        |            |(typeless)    V                       V
        |            +---------> fragment     +------> fixed-function
        |                        program      |(float)   fragment
        |                           |         |             |
        +--------------------------/|/--------+             |
                                    |                       |
                                    | (typeless)            | (float)
                                    V                       |
                                    |<----------------------+
                                    |
                                    +-----------------------+------ ....
                                    |                       |
                                    | (typeless)            | (typeless)
                                    V                       V
                                  color                   color
                                attachment              attachment
                                    0                       1
```

(5) Instructions can operate on signed integer, unsigned integer, and floating-point values. Some operations make sense on all three data types? How is this supported, and what type checking support is provided by the assembler?

RESOLVED: One important property of the instruction set is that the data type for all operands and the result is fully specified by the instructions themselves. For instructions (such as ADD) that make sense for both integer and floating-point values, an optional data type modifier is provided to indicate which type of operation should be performed. For example, "ADD.S", "ADD.U", and "ADD.F", add signed integers, unsigned integers, or floating-point values, respectively. If no data type modifier is provided, ".F" is assumed if the instruction can apply to floating-point values and ".S" is assumed otherwise.

To help identify errors where the wrong data type is used -- for example, adding integer values in an ADD instruction that omits a data type modifier and thus defaults to "ADD.F" -- variables may be declared with optional data type modifiers. In the following code:

```
INT TEMP a;
UINT TEMP b;
FLOAT TEMP c;
TEMP d;
```

"a", "b", "c", and "d" are declared as temporary variables holding signed integer, unsigned integer, floating-point, and typeless values. Since each instruction fully specifies the data type of each operand and its result, these data types can be checked against the data type assigned to the variables operated on. If the types don't match, and the variable is not typeless, an error is reported. The opcode modifier ".NTC" can be used to ignore such errors on a per-opcode basis, if required.

Note that when bindings are used directly in instructions, they are always considered typeless for simplicity. Some fixed-function bindings have an obvious data type, but other bindings (e.g., program parameters) can hold either integer or floating-point values, depending on how they were specified.

Variable data types are optional. Typeless variables are provided because some programs may want to reuse the same variable in several places with different data types.

(6) Should both signed (INT) and unsigned integer (UINT) data types be provided?

RESOLVED: Yes. Signed and unsigned integer operations are supported. Providing both "INT" and "UINT" variable modifiers distinguish between signed and unsigned values for type checking purposes, to ensure that unsigned values aren't read as signed values and vice versa.

This specification says if a value is read a signed integer, but was written as an unsigned integer, the value returned is undefined. However, signed and unsigned integers are interchangeable in practice, except for very large unsigned integers (which can't be represented as signed values of the equivalent size) or negative signed integers.

If programs know that they won't generate negative or very large values, signed and unsigned integers can be used interchangeably. To avoid type errors in the assembler in this case, typeless variables can be used. Or the ".NTC" modifier can be used when appropriate.

(7) Integer and floating-point constants are supported in the instruction set. Integer constants might be interpreted to mean either "real integer" values or floating-point values. How are they supported?

RESOLVED: When an obvious floating point constant is specified (e.g., "3.0"), the developers' intent is clear. If you try to use a floating-point value in an instruction that wants an integer operand, or a declaration of an integer parameter variable, the program will fail to load. An integer constant used in an instruction isn't quite as clear. But its meaning can be easily inferred because the operand types of instructions are well-known at compile time. An integer multiply involving the constant "2" will interpret the "2" as an integer. A floating-point multiply involving the same constant "2" will interpret it as a floating-point value.

The only real problem is for a parameter declaration that is typeless. For typed variables, the intent is clear:

```
INT PARAM two = 2;              # use integer 2
FLOAT PARAM twoPt0 = 2;         # use floating-point 2.0
```

For typeless variables, there's no context to go on:

```
PARAM two = 2;                  # 2?  2.0?
```

This extension is intended to be largely upward-compatible with ARB_vertex_program, ARB_fragment_program, and the other extensions built on top of them. In all of these, the previous declaration is legal and means "2.0". For compatibility, we choose to interpret integer constants in this case as floating-point values. The assembler in the NVIDIA implementation will issue a warning if this case ever occurs.

This extension does not provide decoration of integer constant values -- we considered adding suffixed integers such as "2U" to mean "2, and don't even think about converting me to a float!". We expect that it will be sufficient to use the "INT" or "FLOAT" modifiers to disambiguate effectively.

(8) Should hexadecimal constants (e.g., 0x87A3 or 0xFFFFFFFF) be supported?

RESOLVED: Yes.

(9) Should we provide data type modifiers with explicit component sizes? For example, "INT8", "FLOAT16", or "INT32". If so, should we provide a mechanism to query the size (in bits) of a variable, or of different variable types/qualifiers?

RESOLVED: No.

(10) Should this extension provide better support for array variables?

RESOLVED: Yes; array variables of all types are allowed.

In ARB_vertex_program, program parameter (constant) variables could be addressed as arrays. Temporary variables, vertex attributes, and vertex results could not be declared as arrays.

In NV_vertex_program3 and NV_fragment_program2, relative addressing was supported in program bindings:

```
MOV R0, vertex.attrib[A0.x];            # vertex
MOV result.texcoord[A0.x], R0;          # vertex
MOV R0, fragment.texcoord[A0.x];        # fragment -- inside LOOP
```

Explicitly declared attribute or result arrays were not supported, and temporaries could also not be arrays.

This extension allows users to declare attribute, result, and temporary arrays such as:

```
ATTRIB attribs[] = { vertex.attrib[7..11] };
TEMP scratch[10];
RESULT texcoords[] = { result.texcoord[0..3] };
```

Additionally, the relative addressing mechanisms provided by NV_vertex_program3 and NV_fragment_program2 are NOT supported in this extension -- instead, declared array variables are the only way to get relative addressing. Using declared arrays allows the assembler to identify which attributes will actually be used. An expression like "vertex.texcoord[A0.x]" doesn't identify which texture coordinates are referenced, and the assembler must be conservative in this case and assume that they all are.

(11) Is relative addressing of temporaries allowed?

RESOLVED: Yes. However, arrays of temporaries may end up being stored in off-chip memory, and may be slower to access than non-array temporaries.

(12) Should this extension add bindings to pass generic attributes between vertex, geometry, and fragment programs, or are texture coordinates sufficient?

RESOLVED: While texture coordinates have been used in the past, generic attributes should be provided.

The assembler provides a large set of bindings and automatically eliminates generic attributes or components that are unused. At each interface between programs, there is an implementation-dependent limit on the number of attribute components that can be passed.

There are several reasons that this approach was chosen. First, if the number of attributes that can be passed between program stages exceeds the number of existing texture coordinate sets supported when specifying vertex, a second implementation-dependent number of texture coordinates would need to be exposed to cover the number supported between stages.

Second, the mechanisms described above reduce or eliminate the need to pack attributes into four component vectors. Third, "texture coordinates" that have been historically used for texture lookups don't need to be used to pass values that aren't used this way.

(13) The structured branching support in NV_fragment_program2 provides a REP instruction that says to repeat a block of code <N> times, as well as a LOOP instruction that does the same, but also provides a special loop counter variable. What sort of looping mechanism should we provide here?

RESOLVED: Provide only the REP instruction. The functionality provided by the LOOP instruction can be easily achieved by using an integer temporary as the loop index. This avoids two annoyances of the old LOOP models: (a) the loop index (A0.x) is a special variable name, while all other variables are declared normally and (b) instructions can only access the loop index of the innermost loop -- loop indices at higher nesting levels are not accessible.

One other option was a considered -- a "LOOPV" instruction (LOOP with a variable where the program specified a variable name and component to hold the loop index, instead of using the implicit variable name "A0.x". In the end, it was decided that using an integer temporary as a loop counter was sufficient.

(14) The structured branching support in NV_fragment_program2 provides a REP instruction that requires a loop count. Some looping constructs may not have a definite loop count, such as a "while" statement in C. Should this construct be supported, and if so, how?

RESOLVED: The REP instruction is extended to make the loop count optional. If no loop count is provided, the REP instruction specified a loop that can only be exited using the BRK (break) or RET instructions. To avoid obvious infinite loops, an error will be reported if a REP/ENDREP block contains no BRK instruction at the current nesting level and no RET instruction at any nesting level.

To implement a loop like "while (value < 7.0) ...", code such as the following can be used:

```
TEMP cc;                        # dummy variable
REP;
  SLT.CC cc.x, value.x, 7.0;    # compare value.x to 7.0, set CC0
  BRK NE.x;                     # break out if not true
  ...
  ...                           # presumably update value!
  ...
ENDREP;
```

(15) The structured branching support in NV_fragment_program2 provides a BRK instruction that operates like C's "break" statement. Should we provide something similar to C's "continue" statement, which skips to the next iteration of the loop?

RESOLVED: Yes, a new CONT opcode is provided for this purpose.

(16) Can the BRK or CONT instructions break out of multiple levels of nested loops at once?

RESOLVED: No. BRK and CONT only exit the current nesting level. To break out of multiple levels of nested loops, multiple BRK/CONT instructions are required.

(17) For REP instructions, is the loop counter reloaded on each iteration of the loop?

RESOLVED: No. The loop counter is loaded once at the top of the loop, compared to zero at the top of the loop, and decremented when each loop iteration completes. A program may overwrite the variable used to specify the initial value of the loop counter inside the loop without affecting the number of times the loop body is executed.

(18) How are floating-point values represented in this extension? What about floating-point arithmetic operations?

RESOLVED: In the initial hardware implementation of this extension, floating-point values are represented using the standard 32-bit IEEE single-precision encoding, consisting of a sign bit, 8 exponent bits, and 23 mantissa bits. Special encodings for NaN (not a number), +/-INF (infinity), and positive and negative zero are supported. Denorms (values less than 2^-126, which have an exponent encoding of "0" and no implied leading one) are supported, but may be flushed to zero, preserving the sign bit of the original value. Arithmetic operations are carried out at single-precision using normal IEEE floating-point rules, including special rules for generating infinities, NaNs, and zeros of each sign.

Floating-point temporaries declared as "SHORT" may be, but are not necessarily, stored as 16-bit "fp16" values (sign bit, five exponent bits, ten mantissa bits), as specified in the NV_float_buffer and ARB_half_float_pixel extensions.

(19) Should we provide a method to declare how fragment attributes are interpolated? It is possible to have flat-shaded attributes, perspective-corrected attributes, and centroid-sampled attributes.

RESOLVED: Yes. Fragment program attribute variable declarations may specify the "FLAT", "NOPERSPECTIVE", and "CENTROID" modifiers.

These modifiers are documented in detail in the NV_fragment_program4 specification.

(20) Should vertex and primitive identifiers be supported? If so, how?

RESOLVED: A vertex identifier is available as "vertex.id" in a vertex program. The vertex ID is equal to value effectively passed to ArrayElement when the vertex is specified, and is defined only if vertex arrays are used with buffer objects (VBOs).

A primitive identifier is available as "primitive.id" in a geometry or fragment program. The primitive ID is equal to the number of primitives processed since the last implicit or explicit call to glBegin().

See the NV_vertex_program4 spec for more information on vertex IDs, and the NV_geometry_program4 or NV_fragment_program4 specs for more information on primitive IDs.

(21) For integer opcodes, should a bitwise inversion operator "~" be provided, analogous to existing negation operator?

RESOLVED: No. If this operator were provided, it might allow a program to evaluate the expression "a&(~b)" using a single instruction:

```
AND.U a, a, ~b;
```

Instead, it is necessary to instead do something like:

```
UINT TEMP t;
NOT.U t, b;
AND.U a, a, t;
```

If necessary, this functionality could be added in a subsequent extension.

(22) What happens if you negate or take the absolute value of the biggest-magnitude negative integer?

RESOLVED: Signed integers are represented using two's complement representation. For 32-bit integers, the largest possible value is 2^31-1; the smallest possible value is -2^31. There is no way to represent 2^31, which is what these operators "should" return. The value returned in this case is the original value of -2^31.

(23) How do condition codes work? How are they different from those provided in previous NVIDIA extensions?

RESOLVED: There are two condition codes -- CC0 and CC1 -- each of which is a four-component vector. The condition codes are set based on the result of an instruction that specifies a condition code update modifier. Examples include:

```
ADD.S.CC  R0, R1, R2;       # add signed integers R1 and R2, update
                            #   CC0 based on the result, write the
                            #   final value to R0
ADD.F.CC1 R3, R4, R5;       # add floats R4 and R5, update CC1 based
                            #   on the result, write the final value
                            #   to R3
ADD.U.CC0 R6.xy, R7, R8;    # add unsigned integers R7 and R8, update
                            #   CC0 (x and y components) based on the
                            #   result, write the final value to R6
                            #   (x and y components)
```

Condition codes can be used for conditional writes, conditional branches, or other operations. The condition codes aren't used directly, but are instead used with a condition code test such as "LT" (less than) or "EQ" (equal to). Examples include:

```
  MOV R0 (GT.x), R1;          # move R1 to R0 only if the x component of
                              #   CC0 indicates a result of ">0"
  MOV R2 (NE1), R3;           # component-wise move of R3 to R2 if the
                              #   corresponding component of CC1
                              #   indicates a result of "!=0"
  IF LE0.xyxy;                # execute the block of code if the x or
    ...                       #   y components of CC0 indicate a result
  ENDIF;                      #   of "<=0"
  REP;
    ...
    BRK EQ1.xyzx;             # break out of loop if the x, y, or z
  ENDREP;                     #   components of CC1 indicate a result of
                              #   "==0".
```

Previous NVIDIA extensions provide eight tests, which are still supported here. The tests "EQ" (equal), "GE" (greater/equal), "GT" (greater than), "LE" (less/equal), "LT" (less than), and "NE" (not equal) can be used to determine the relation of the result used to set the condition code with zero. The tests "TR" (true) and "FL" (false), are special tests that always evaluate to true or false respectively.

For floating-point results, a NaN (not a number) encoding causes the "NE" condition to evaluate to TRUE and all other conditions to evaluate to FALSE. IEEE encodings for "negative" and "positive" zero are both treated as equal to zero.

Condition codes are implemented as a set of flags, which are set depending on the type of operation, as described in the spec.

For instructions that return floating-point or signed integer values, the normal condition code tests reliably indicate the relationship of the result to zero. For instructions that return unsigned values, the condition codes are a bit more complicated. For example, the sign flag is set if the most significant bit of the result written is set. As a result, very large unsigned integer values (e.g., 0x80000000 - 0xFFFFFFFF) are effectively treated as negative values. Condition code tests should be used with care with unsigned results -- to test if an unsigned integer is ">0", use a sequence like:

```
  MOV.U.CC R0, R1;            # move R1 to R0, set condition code
  IF NE;                      # test if the result is "!=0", a very
    ...                       #   large value might fail "GT"!
  ENDIF;
```

This extension provides a number of additional condition code tests useful for different floating-point or integer operations:

* NAN (not a number) is true if a floating-point result is a NaN. LEG (less, equal to, or greater) is the opposite of NAN.

* CF (carry flag) is true if an unsigned add overflows, or if an unsigned subtract produces a non-negative value. NCF (no carry flag) is the opposite of CF.

* OF (overflow flag) is true if a signed add or subtract overflows. NOF (no overflow flag) is the opposite of OF.

* SF (sign flag) is true if the sign flag is set. NSF (no sign flag) is the opposite of SF.

* AB (above) is true if an unsigned subtract produces a positive result. BLE (below or equal) is the opposite of AB, and is true if an unsigned subtract produces a negative result or zero. Note that CF can be used to test if the result is greater than or equal to zero, and NCF can be used to test if the result is less than zero.

(24) How do the "set on" instructions (SEQ, SGE, SGT, SLE, SLT, SNE) work with integer values and/or condition codes?

RESOLVED: "Set on" instructions comparing signed and unsigned values return zero if the condition is false, and an integer with all bits set if the condition is true. If the result is signed, it is interpreted as -1. If the result is unsigned, it is interpreted the largest unsigned value (0xFFFFFFFF for 32-bit integers). This is different from the floating-point "set on", which is defined to return 1.0.

This specific result encoding was chosen so that bitwise operators (NOT, AND, OR, XOR) can be used to evaluate boolean expressions.

When performing condition code tests on the results of an integer "set on" instruction, keep in mind that a TRUE result has the most significant bit set and will be interpreted as a negative value. To test if a condition is true, use "NE" (!=0). A condition code test of "GT" will always fail if the condition code was written by an integer "set on" instruction.

(25) What new texture functionality is provided?

RESOLVED: Several new features are provided.

First, the TXF (texel fetch) instruction allows programs to access a texture map like a normal array. Integer coordinates identifying an individual texel and LOD are provided, and the corresponding texture data is returned without filtering of any type.

Second, the TXQ (texture size query) instruction allows programs to query the size of a specified level of detail of a texture. This feature allows programs to perform computations dependent on the size of the texture without having to pass the size as a program parameter or via some other mechanism.

Third, applications may specify a constant texel offset in a texture instruction that moves the texture sample point by the specified number of texels. This offset can be used to perform custom texture filtering, and is also independent of the size of the texture LOD -- the same offsets are applied, regardless of the mipmap level.

Fourth, shadow mapping is supported for cube map textures. The first three coordinates are the normal (s,t,r) coordinates for a cube map texture lookup, and the fourth component is a depth reference value that can be compared to the depth value stored in the texture.

(26) What "consistency" requirements are in effect for textures accessed via the TXF (texel fetch) instruction?

UNRESOLVED: The texture must be usable for regular texture mapping operations -- if texture sizes or formats are inconsistent and a mipmapped min filter is used, the results are undefined.

(27) How does the TXF instruction work with bordered textures?

RESOLVED: The entire image can be accessed, including the border texels. For a 64x64 2D texture plus border (66x66 overall), the lower left border texel is accessed using the coordinates (-1,-1); the upper right border texel is accessed using the coordinates (64,64).

(28) What should TXQ (texture size query) return for "irrelevant" texture sizes (e.g., height of a 1D texture)? Should it return any other information at the same time?

RESOLVED: This specification leaves all "extra" components undefined.

(29) How do texture offsets interact with cubemap textures?

RESOLVED: They are not supported in this extension.

(30) How do texture offsets interact with mipmapped textures?

RESOLVED: The texture offsets are added after the (s,t,r) coordinates have been divided by q (if applicable) and converted to (u,v,w) coordinates by multiplying by the size of the selected texture level. The offsets are added to the (u,v,w) coordinates, and always move the sample point by an integral number of texel coordinates. If multiple mipmaps are accessed, the sample point in each mipmap level is moved by an identical offset. The applied offsets are independent of the selected mipmap level.

(31) How do shadow cube maps work?

UNRESOLVED: An application can define a cube map texture with a DEPTH_COMPONENT internal format, and then render a scene using the cube map faces as the depth buffer(s). When rendering the projection should be set up using the "center" of the cubemap as the eye, and using a normal projection matrix. When applying the shadow map, the fragment program read the (x,y,z) eye coordinates, compute the length of the major axis (MAX(|x|,|y|,|z|) and then transform this coordinate to [0,1] space using the same parameters used to derive Z in the projection matrix. A 4-component vector consisting of x, y, z, and this computed depth value should be passed to the texture lookup, and normal shadow mapping operations will be performed.

This issue should include the math needed to do this computation and sample code.

(32) Integer multiplies can overflow by a lot. Should there be some way to return the high part of both unsigned and signed integer multiplies?

RESOLVED: Yes. The ".HI" multipler is provided to do a return the 32 MSBs of a 32x32 integer multiply. The instruction sequence:

```
INT TEMP R0, R1, R2, R3;
MUL.S    R0, R2, R3;
MUL.S.HI R1, R2, R3;
```

will do a 32x32 signed integer multiply of R2 and R3, with the 32 LSBs of the 64-bit result in R0 and the 32 MSBs in R1.

(33) Should there be any other special multiplication modifiers?

RESOLVED: Yes. The ".S24" and ".U24" modifiers allow for signed and unsigned integer multiplies where both operands are guaranteed to fit in the least significant 24 bits. On some architectures supporting this extension, ".S24" and ".U24" integer multiplies may be faster than general-purpose ".S" and ".U" multiplies. If either value doesn't fit in 24 bits, the results of the operation are undefined -- implementations may, but are not required to, ignore the MSBs of the operands if ".S24" or ".U24" is specified.

(34) This extension provides subroutines, but doesn't provide a stack to push and pop parameters. How do we deal with this? NV_vertex_program3 supported PUSHA/POPA instructions to push and pop address registers.

RESOLVED: No explicit stack is required. A program can implement a stack by allocating a temporary array plus a single integer temporary to use as the stack "pointer". For example:

```
TEMP stack[256];               # 256 4-component vectors
INT TEMP sp;                   # sp.x == stack pointer
INT TEMP cc;                   # condition code results

function:
  SGE.S.CC cc.x, sp.x, 256;    # compute stackPointer >= 256
  RET NE.x;                    # return if TRUE
  MOV stack[sp], R0;           # push R0 onto the stack
  ADD.S sp.x, sp.x, 1;
  ...
  SUB.S sp.x, sp.x, 1;         # pop R0 off the stack
  MOV R0, stack[sp];
  RET
```

(35) Should we provide new vector semantics for previously-defined opcodes (e.g., LG2 computes a component-wise logarithm)?

RESOLVED: Not in this extension. The instructions we define here are compatible with the vector or scalar nature of previously defined opcodes. This simplifies the implementation of an assembler that needs to support both old and new instruction sets.

(36) Should it really be undefined to read from a register storing data of one type with an instruction of the other type (e.g., to read the bits of a floating-point number as an unsigned integer)?

RESOLVED: The spec describes undefined results for simplicity. In practice, mixing data types can be done, where signed integers are represented as two's complement integers and floating-point numbers are represented using IEEE single-precision representation. For example:

```
  TEMP R0, R1;                  # typeless
  MOV.U R0, 0x3F800000;         # R0 = 1.0
  MOV.U R1, 0xBF800000;         # R1 = -1.0
  MUL.F R0, R0, R1;             # R0 = -1 * 1 = -1 (0xBF800000)
  XOR.U R0, R0, R1;             # R0 = 0xBF800000 ^ 0xBF800000 = 0
  NOT.U R0, R0;                 # R0 = 0xFFFFFFFF
  I2F.S R0, R0;                 # R0 = -1.0 (0xFFFFFFFF = -1 signed)
  SEQ.F R0, R0, R1;             # R0 = 1.0 (-1.0 == -1.0)
```

(37) Buffer objects can be sourced as program parameters using the NV_parameter_buffer_object extension. How are they accessed in a program?

RESOLVED: The instruction set and existing program environment and local parameter bindings operate largely on four-component vectors. However, NV_parameter_buffer_object exposes the ability to reach into buffers consisting of user-generated data or data written to the buffer object by the GPU. Such data sets may not consist entirely four-component floating-point vectors, so a four-component vector API may be unnatural. An application might need to reformat its data set to deal with this issue. Or it might generate odd code to compensate for mis-alignment -- for example, reading an array of 3-component vectors by doing two four-component vector accesses and then rotating based on alignment. Neither approach is particularly satisfying.

Instead, this extension takes the approach of treating parameter buffers as array of scalar words. When an individual buffer element is read, the single word is replicated to produce a four-component vector. To access an array of 3-component vectors, code like the following can be used:

```
  PARAM buffer[] = { program.buffer[0] };
  INT TEMP index;
  TEMP R0;
  ...
  MUL.S index, index, 3;        # to read "vec3" #X, compute 3*X
  MOV R0.x, buffer[index+0];
  MOV R0.y, buffer[index+1];
  MOV R0.z, buffer[index+2];
```

(38) Should recursion be allowed? If so, how is the total amount of recursion limited?

RESOLVED: Recursion is allowed, and a call stack is provided by the implementation. The size of the call stack is limited to the implementation-dependent constant MAX_PROGRAM_CALL_DEPTH, and when a the call stack is full, the results of further CAL instructions is undefined. In the initial implementation of this extension, such instructions will have no effect.

Note that no stack is provided to hold local registers; a program may implement its own via a temporary array and integer stack "pointer".

(39) Variables are all four-component vectors in previous extensions. Should scalar or small-vector variables be provided?

RESOLVED: It would be a useful feature, but it was left out for simplicity. In practice, a variable where only the X component is used will be equivalent to a scalar.

(40) The PK (pack) and UP* (unpack) instructions allow packing multiple components of data into a single component. The bit packing is well-defined. Should we require specific data types (e.g., unsigned integer) to hold packed values?*

RESOLVED: No. Previous instruction sets only allowed programs to write packed values to a floating-point variable (the only data type provided). We will allow packed results to be written to a variable of any data type. Integer instructions can be used to manipulate bits of packed data in place.

(41) What happens when converting integers to floats or vice versa if there is insufficient precision or range to represent the result?

RESOLVED: For integer-to-float conversions, the nearest representable floating-point value is used, and the least significant bits of the original integer value are lost. For float-to-integer conversions, out-of-range values are clamped to the nearest representable integer.

(42) Why are some of the grammar rules so bizarre (e.g., attribUseD, attribUseV, attribUseS, attribUseVNS)?

RESOLVED: This grammar is based upon the original ARB_vertex_program grammar, which has a number of "interesting" characteristics. For example, some of the bindings provided by ARB_vertex_program naturally require some amount of lookahead. For example, a vertex program can write an output color using any of the following:

```
MOV result.color, 0;            # primary color
MOV result.color.primary, 0;    # primary color again
MOV result.color.secondary, 0;  # secondary color this time
```

The pieces of the color binding are separated by "." tokens. However, writemasks are also supported, which also use "." before the write mask. So, we could also have something like:

```
MOV result.color.xyz, 0;        # primary color with W masked off
```

In this form, a parser needs to look at both the "." and the "xyz" to determine that the binding being used is "result.color" (and not "result.color.secondary").

Additionally, some checks that should probably be semantic errors (e.g., allowing different swizzle or scalar operand selectors per instruction, or disallowing both in the case of SWZ) we specified in the original grammar.

ARB_fragment_program and subsequent NVIDIA instructions built upon this, and the grammar for this extension was rewritten in the current form so it could be validated more easily.

(43) This is an NV extension (NV_gpu_program4). Why does the MAX_PROGRAM_TEXEL_OFFSET_EXT token has an "EXT" suffix?

RESOLVED: This token is shared between this extension and the comparable high-level GLSL programmability extension (EXT_gpu_shader4). Rather than provide a duplicate set of token names, we simply use the EXT version here.

Revision History

Rev.	Date	Author	Changes
4	02/04/08	pbrown	Fix errors in texture wrap mode handling. Added a missing clamp to avoid sampling border in REPEAT mode. Fixed incorrectly specified weights for LINEAR filtering.

Name

    NV_half_float

Name Strings

    GL_NV_half_float

Notice

IP Status

    NVIDIA Proprietary.

Status

    Implemented in CineFX (NV30) Emulation driver, August 2002.
    Shipping in Release 40 NVIDIA driver for CineFX hardware, January 2003.

Version

```
    Last Modified Date:         02/25/2004
    NVIDIA Revision:            9
```

Number

    283

Dependencies

    Written based on the wording of the OpenGL 1.3 specification.

    OpenGL 1.1 is required.

    NV_float_buffer affects the definition of this extension.

    EXT_fog_coord affects the definition of this extension.

    EXT_secondary_color affects the definition of this extension.

    EXT_vertex_weighting affects the definition of this extension.

    NV_vertex_program affects the definition of this extension.

Overview

    This extension introduces a new storage format and data type for
    half-precision (16-bit) floating-point quantities. The floating-point
    format is very similar to the IEEE single-precision floating-point
    standard, except that it has only 5 exponent bits and 10 mantissa bits.
    Half-precision floats are smaller than full precision floats and provide a
    larger dynamic range than similarly-sized normalized scalar data types.

    This extension allows applications to use half-precision floating point
    data when specifying vertices or pixel data. It adds new commands to

specify vertex attributes using the new data type, and extends the existing vertex array and image specification commands to accept the new data type.

This storage format is also used to represent 16-bit components in the floating-point frame buffers, as defined in the NV_float_buffer extension.

Issues

What should the new data type be called? "half"? "hfloat"? In addition, what should the immediate mode function suffix be? "h"? "hf"?

RESOLVED: half and "h". This convention builds on the convention of using the type "double" to describe double-precision floating-point numbers. Here, "half" will refer to half-precision floating-point numbers.

Even though the 16-bit float data type is a first-class data type, it is still more problematic than the other types in the sense that no native programming languages support the data type. "hfloat/hf" would have reflected a second-class status better than "half/h".

Both names are not without conflicting precedents. The name "half" is used to connote 16-bit scalar values on some 32-bit CPU architectures (e.g., PowerPC). The name "hfloat" has been used to describe 128-bit floating-point data on VAX systems.

Should we provide immediate-mode entry points for half-precision floating-point data types?

RESOLVED: Yes, for orthogonality. Also useful as a fallback for the "general" case for ArrayElement.

Should we support half-precision floating-point color index data?

RESOLVED: No.

Should half-precision data be accepted by all commands that accept pixel data or only a subset?

RESOLVED: All functions. Note that some textures or frame buffers may store the half-precision floating-point data natively.

Since half float data would be accepted in some cases, it will be necessary for drivers to provide some data conversion code. This code can be reused to handle the less common commands.

New Procedures and Functions

```
    void Vertex2hNV(half x, half y);
    void Vertex2hvNV(const half *v);
    void Vertex3hNV(half x, half y, half z);
    void Vertex3hvNV(const half *v);
    void Vertex4hNV(half x, half y, half z, half w);
    void Vertex4hvNV(const half *v);
    void Normal3hNV(half nx, half ny, half nz);
    void Normal3hvNV(const half *v);
    void Color3hNV(half red, half green, half blue);
    void Color3hvNV(const half *v);
    void Color4hNV(half red, half green, half blue, half alpha);
    void Color4hvNV(const half *v);
    void TexCoord1hNV(half s);
    void TexCoord1hvNV(const half *v);
    void TexCoord2hNV(half s, half t);
    void TexCoord2hvNV(const half *v);
    void TexCoord3hNV(half s, half t, half r);
    void TexCoord3hvNV(const half *v);
    void TexCoord4hNV(half s, half t, half r, half q);
    void TexCoord4hvNV(const half *v);
    void MultiTexCoord1hNV(enum target, half s);
    void MultiTexCoord1hvNV(enum target, const half *v);
    void MultiTexCoord2hNV(enum target, half s, half t);
    void MultiTexCoord2hvNV(enum target, const half *v);
    void MultiTexCoord3hNV(enum target, half s, half t, half r);
    void MultiTexCoord3hvNV(enum target, const half *v);
    void MultiTexCoord4hNV(enum target, half s, half t, half r, half q);
    void MultiTexCoord4hvNV(enum target, const half *v);
    void FogCoordhNV(half fog);
    void FogCoordhvNV(const half *fog);
    void SecondaryColor3hNV(half red, half green, half blue);
    void SecondaryColor3hvNV(const half *v);
    void VertexWeighthNV(half weight);
    void VertexWeighthvNV(const half *weight);
    void VertexAttrib1hNV(uint index, half x);
    void VertexAttrib1hvNV(uint index, const half *v);
    void VertexAttrib2hNV(uint index, half x, half y);
    void VertexAttrib2hvNV(uint index, const half *v);
    void VertexAttrib3hNV(uint index, half x, half y, half z);
    void VertexAttrib3hvNV(uint index, const half *v);
    void VertexAttrib4hNV(uint index, half x, half y, half z, half w);
    void VertexAttrib4hvNV(uint index, const half *v);
    void VertexAttribs1hvNV(uint index, sizei n, const half *v);
    void VertexAttribs2hvNV(uint index, sizei n, const half *v);
    void VertexAttribs3hvNV(uint index, sizei n, const half *v);
    void VertexAttribs4hvNV(uint index, sizei n, const half *v);
```

New Tokens

    Accepted by the <type> argument of VertexPointer, NormalPointer,
    ColorPointer, TexCoordPointer, FogCoordPointerEXT,
    SecondaryColorPointerEXT, VertexWeightPointerEXT, VertexAttribPointerNV,
    DrawPixels, ReadPixels, TexImage1D, TexImage2D, TexImage3D, TexSubImage1D,
    TexSubImage2D, TexSubImage3D, and GetTexImage:

```
        HALF_FLOAT_NV                                   0x140B
```

Additions to Chapter 2 of the OpenGL 1.3 Specification (OpenGL Operation)

Modify Section 2.3, GL Command Syntax (p. 7)

(Modify the last paragraph, p. 7. In the text below, "e*" represents the epsilon character used to indicate no character.)

These examples show the ANSI C declarations for these commands. In general, a command declaration has the form

```
        rtype Name{e*1234}{e* b s i h f d ub us ui}{e*v}
          ( [args ,] T arg1, ... , T argN [, args]);
```

(Modify Table 2.1, p. 8 -- add new row)

Letter	Corresponding GL Type
h	half

(add after last paragraph, p. 8) The half data type is a floating-point data type encoded in an unsigned scalar data type. If the unsigned scalar holding a half has a value of N, the corresponding floating point number is

```
        (-1)^S * 0.0,                         if E == 0 and M == 0,
        (-1)^S * 2^-14 * (M / 2^10),          if E == 0 and M != 0,
        (-1)^S * 2^(E-15) * (1 + M/2^10),     if 0 < E < 31,
        (-1)^S * INF,                         if E == 31 and M == 0, or
        NaN,                                  if E == 31 and M != 0,
```

where

```
        S = floor((N mod 65536) / 32768),
        E = floor((N mod 32768) / 1024), and
        M = N mod 1024.
```

INF (Infinity) is a special representation indicating numerical overflow. NaN (Not a Number) is a special representation indicating the result of illegal arithmetic operations, such as computing the square root or logarithm of a negative number. Note that all normal values, zero, and INF have an associated sign. -0.0 and +0.0 are considered equivalent for the purposes of comparisons. Note also that half is not a native type in most CPUs, so some special processing may be required to generate or interpret half data.

(Modify Table 2.2, p. 9 -- add new row)

```
                Minimum
    GL Type    Bit Width    Description
    -------    ---------    -----------------------------------
    half          16        half-precision floating-point value
                            encoded in an unsigned scalar
```

Modify Section 2.7, Vertex Specification, p. 19

(Modify the descriptions of the immediate mode functions in this section,
 including those introduced by extensions.)

```
    void Vertex[234][sihfd]( T coords );
    void Vertex[234][sihfd]v( T coords );
...
    void TexCoord[1234][sihfd]( T coords );
    void TexCoord[1234][sihfd]v( T coords );
...
    void MultiTexCoord[1234][sihfd](enum texture, T coords);
    void MultiTexCoord[1234][sihfd]v(enum texture, T coords);
...
    void Normal3[bsihfd][ T coords );
    void Normal3[bsihfd]v( T coords );
...
    void Color[34][bsihfd ubusui]( T components );
    void Color[34][bsihfd ubusui]v( T components );
...
    void FogCoord[fd]EXT(T fog);
    void FogCoordhNV(T fog);
    void FogCoord[fd]vEXT(T fog);
    void FogCoordhvNV(T fog);
...
    void SecondaryColor3[bsihfd ubusui]( T components );
    void SecondaryColor3hNV( T components );
    void SecondaryColor3[bsihfd ubusui]v( T components );
    void SecondaryColor3hvNV( T components );
...
    void VertexWeightfEXT(T weight);
    void VertexWeighthNV(T weight);
    void VertexWeightfvEXT(T weight);
    void VertexWeighthvNV(T weight);
...
    void VertexAttrib[1234][shfd]NV(uint index, T components);
    void VertexAttrib4ubNV(uint index, T components);
    void VertexAttrib[1234][shfd]vNV(uint index, T components);
    void VertexAttrib4ubvNV(uint index, T components);
    void VertexAttribs[1234][shfd]vNV(uint index, sizei n, T components);
    void VertexAttribs4ubvNV(uint index, sizei n, T components);
....
```

Modify Section 2.8, Vertex Arrays, p. 21

(Modify 1st paragraph on p. 22) ... For <type>, the values BYTE, SHORT,
INT, FLOAT, HALF_FLOAT_NV, and DOUBLE indicate types byte, short, int,
float, half, and double, respectively. ...

(Modify Table 2.4, p. 23)

Command	Sizes	Types
VertexPointer	2,3,4	short, int, float, half, double
NormalPointer	3	byte, short, int, float, half, double
ColorPointer	3,4	byte, ubyte, short, ushort, int, uint, float, half, double
IndexPointer	1	ubyte, short, int, float, double
TexCoordPointer	1,2,3,4	short, int, float, half, double
EdgeFlagPointer	1	boolean
FogCoordPointerEXT	1	float, half, double
SecondaryColorPointerEXT	3	byte, ubyte, short, ushort, int, uint, float, half, double
VertexWeightPointerEXT	1	float, half

Table 2.4: Vertex array sizes (values per vertex) and data types.

Modify Section 2.13, Colors and Coloring, p.44

(Modify Table 2.6, p. 45) Add new row to the table:

GL Type	Conversion
half	c

Modify NV_vertex_program_spec, Section 2.14.3, Vertex Arrays for Vertex Attributes.

(modify paragraph describing VertexAttribPointer) ... type specifies the data type of the values stored in the array. type must be one of SHORT, FLOAT, HALF_FLOAT_NV, DOUBLE, or UNSIGNED_BYTE and these values correspond to the array types short, int, float, half, double, and ubyte respectively. ...

(add to end of paragraph describing mapping of vertex arrays to immediate-mode functions) ... For each vertex attribute, the corresponding command is VertexAttrib[size][type]v, where size is one of [1,2,3,4], and type is one of [s,f,h,d,ub], corresponding to the array types short, int, float, half, double, and ubyte respectively.

Additions to Chapter 3 of the OpenGL 1.3 Specification (Rasterization)

Modify Section 3.6.4, Rasterization of Pixel Rectangles (p. 91)

(Modify Table 3.5, p. 94 -- add new row)

type Parameter Token Name	Corresponding GL Data Type	Special Interpretation
HALF_FLOAT_NV	half	No

Additions to Chapter 4 of the OpenGL 1.3 Specification (Per-Fragment Operations and the Frame Buffer)

Modify Section 4.3.2, Reading Pixels (p. 173)

(modify Final Conversion, p. 177) For an index, if the type is not FLOAT or HALF_FLOAT_NV, final conversion consists of masking the index with the value given in Table 4.6; if the type is FLOAT or HALF_FLOAT_NV, then the integer index is converted to a GL float or half data value. For an RGBA color, components are clamped depending on the data type of the buffer being read. For fixed-point buffers, each component is clamped to [0.1]. For floating-point buffers, if <type> is not FLOAT or HALF_FLOAT_NV, each component is clamped to [0,1] if <type> is unsigned or [-1,1] if <type> is signed and then converted according to Table 4.7.

(Modify Table 4.7, p. 178 -- add new row)

type Parameter	GL Data Type	Component Conversion Formula
HALF_FLOAT_NV	half	c = f

Additions to Chapter 5 of the OpenGL 1.3 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 1.3 Specification (State and State Requests)

None.

Additions to Appendix A of the OpenGL 1.3 Specification (Invariance)

None.

Additions to the AGL/GLX/WGL Specifications

None.

GLX Protocol (Modification to the GLX 1.3 Protocol Encoding Specification)

Add to Section 1.4 (p.2), Common Types

FLOAT16	A 16-bit floating-point value in the format specified in the NV_half_float extension specification.

Modify Section 2.3.3 (p. 79), GL Rendering Commands

The following rendering commands are sent to the server as part of a glXRender request:

```
**Vertex2hvNV**
   2           8               rendering command length
   2           4240            rendering command opcode
   2           FLOAT16         v[0]
   2           FLOAT16         v[1]
```

```
Vertex3hvNV
    2           12              rendering command length
    2           4241            rendering command opcode
    2           FLOAT16         v[0]
    2           FLOAT16         v[1]
    2           FLOAT16         v[2]
    2                           unused

Vertex4hvNV
    2           12              rendering command length
    2           4242            rendering command opcode
    2           FLOAT16         v[0]
    2           FLOAT16         v[1]
    2           FLOAT16         v[2]
    2           FLOAT16         v[3]

Normal3hvNV
    2           12              rendering command length
    2           4243            rendering command opcode
    2           FLOAT16         v[0]
    2           FLOAT16         v[1]
    2           FLOAT16         v[2]
    2                           unused

Color3hvNV
    2           12              rendering command length
    2           4244            rendering command opcode
    2           FLOAT16         v[0]
    2           FLOAT16         v[1]
    2           FLOAT16         v[2]
    2                           unused

Color4hvNV
    2           12              rendering command length
    2           4245            rendering command opcode
    2           FLOAT16         v[0]
    2           FLOAT16         v[1]
    2           FLOAT16         v[2]
    2           FLOAT16         v[3]

TexCoord1hvNV
    2           8               rendering command length
    2           4246            rendering command opcode
    2           FLOAT16         v[0]
    2                           unused

TexCoord2hvNV
    2           8               rendering command length
    2           4247            rendering command opcode
    2           FLOAT16         v[0]
    2           FLOAT16         v[1]
```

```
        **TexCoord3hvNV**
            2           12              rendering command length
            2           4248            rendering command opcode
            2           FLOAT16         v[0]
            2           FLOAT16         v[1]
            2           FLOAT16         v[2]
            2                           unused

        TexCoord4hvNV
            2           12              rendering command length
            2           4249            rendering command opcode
            2           FLOAT16         v[0]
            2           FLOAT16         v[1]
            2           FLOAT16         v[2]
            2           FLOAT16         v[3]

        MultiTexCoord1hvNV
            2           12              rendering command length
            2           4250            rendering command opcode
            4           ENUM            target
            2           FLOAT16         v[0]
            2                           unused

        MultiTexCoord2hvNV
            2           12              rendering command length
            2           4251            rendering command opcode
            4           ENUM            target
            2           FLOAT16         v[0]
            2           FLOAT16         v[1]

        MultiTexCoord3hvNV
            2           16              rendering command length
            2           4252            rendering command opcode
            4           ENUM            target
            2           FLOAT16         v[0]
            2           FLOAT16         v[1]
            2           FLOAT16         v[2]
            2                           unused

        MultiTexCoord4hvNV
            2           16              rendering command length
            2           4253            rendering command opcode
            4           ENUM            target
            2           FLOAT16         v[0]
            2           FLOAT16         v[1]
            2           FLOAT16         v[2]
            2           FLOAT16         v[3]

        FogCoordhvNV
            2           8               rendering command length
            2           4254            rendering command opcode
            2           FLOAT16         v[0]
            2                           unused
        SecondaryColor3hvNV
            2           12              rendering command length
            2           4255            rendering command opcode
            2           FLOAT16         v[0]
            2           FLOAT16         v[1]
            2           FLOAT16         v[2]
            2                           unused

        VertexWeighthvNV
            2           8               rendering command length
            2           4256            rendering command opcode
            2           FLOAT16         v[0]
            2                           unused

        VertexAttrib1hvNV
            2           12              rendering command length
            2           4257            rendering command opcode
            4           CARD32          index
            2           FLOAT16         v[0]
            2                           unused

        VertexAttrib2hvNV
            2           12              rendering command length
            2           4258            rendering command opcode
            4           CARD32          index
            2           FLOAT16         v[0]
            2           FLOAT16         v[1]

        VertexAttrib3hvNV
            2           16              rendering command length
            2           4259            rendering command opcode
            4           CARD32          index
            2           FLOAT16         v[0]
            2           FLOAT16         v[1]
            2           FLOAT16         v[2]
            2                           unused

        VertexAttrib4hvNV
            2           16              rendering command length
            2           4260            rendering command opcode
            4           CARD32          index
            2           FLOAT16         v[0]
            2           FLOAT16         v[1]
            2           FLOAT16         v[2]
            2           FLOAT16         v[3]

        VertexAttribs1hvNV
            2           12+2*n+p        rendering command length
            2           4261            rendering command opcode
            4           CARD32          index
            4           CARD32          n
            2*n         LISTofFLOAT16   v
            p                           unused, p=pad(2*n)
        VertexAttribs2hvNV
           2           12+4*n          rendering command length
           2           4262            rendering command opcode
           4           CARD32          index
           4           CARD32          n
           4*n         LISTofFLOAT16   v

        VertexAttribs3hvNV
           2           12+6*n+p        rendering command length
           2           4263            rendering command opcode
           4           CARD32          index
           4           CARD32          n
           6*n         LISTofFLOAT16   v
           p                           unused, p=pad(6*n)

        VertexAttribs4hvNV
           2           12+8*n          rendering command length
           2           4264            rendering command opcode
           4           CARD32          index
           4           CARD32          n
           8*n         LISTofFLOAT16   v
```

Modify Section 2.3.4, GL Rendering Commands That May Be Large (p. 127)

(Modify the ARRAY_INFO portion of the DrawArrays encoding (p.129) to reflect the new data type supported by vertex arrays.)

```
        ARRAY_INFO

           4       enum                    data type
                   0x1400   i=1            BYTE
                   0x1401   i=1            UNSIGNED_BYTE
                   0x1402   i=2            SHORT
                   ...
                   0x140B   i=2            HALF_FLOAT_NV
           4       INT32                   j
           4       ENUM                    array type
               ...
```

Modify Appendix A, Pixel Data (p. 148)

(Modify Table A.1, p. 149 -- add new row for HALF_FLOAT_NV data)

type	Encoding	Protocol Type	nbytes
HALF_FLOAT_NV	0x140B	CARD16	2

Dependencies on NV_float_buffer

If NV_float_buffer is not supported, the fixed and floating-point color buffer language in ReadPixels "Final Conversion" should be removed.

Dependencies on EXT_fog_coord, EXT_secondary_color, and EXT_vertex_weighting

If EXT_fog_coord, EXT_secondary_color, or EXT_vertex_weighting are not supported, references to FogCoordPointerEXT, SecondaryColorPointerEXT, and VertexWeightEXT, respectively, should be removed.

Dependencies on NV_vertex_program

If NV_vertex_program is not supported, references to VertexAttribPointerNV should be removed, as should references to VertexAttrib*h[v] commands.

Errors

None.

New State

None.

New Implementation Dependent State

```
Rev.    Date    Author   Changes
----  -------- --------  --------------------------------------------
  9   02/25/04  pbrown   Fixed incorrect language using division by zero
                         as an example of something producing a NaN.
```

Name

    NV_light_max_exponent

Name Strings

    GL_NV_light_max_exponent

Notice

Version

May 20, 1999

Number

189

Dependencies

None

Overview

Default OpenGL does not permit a shininess or spot exponent over 128.0. This extension permits implementations to support and advertise a maximum shininess and spot exponent beyond 128.0.

Note that extremely high exponents for shininess and/or spot light cutoff will require sufficiently high tessellation for acceptable lighting results.

Paul Deifenbach's thesis suggests that higher exponents are necessary to approximate BRDFs with per-vertex ligthing and multiple passes.

New Procedures and Functions

None

New Tokens

Accepted by the <pname> parameters of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

```
        MAX_SHININESS_NV                  0x8504
        MAX_SPOT_EXPONENT_NV              0x8505
```

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

In Table 2.7, change the srm range entry to read:

"(range: [0.0, value of MAX_SHININESS_NV])"

In Table 2.7, change the srli range entry to read:

"(range: [0.0, value of MAX_SPOT_EXPONENT_NV])"

Add to the end of the second paragraph in Section 2.13.2:

"The values of MAX_SHININESS_NV and MAX_SPOT_EXPONENT_NV are implementation dependent, but must be equal or greater than 128."

Additions to Chapter 3 of the GL Specification (Rasterization)

None

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

None.

Additions to Chapter 5 of the GL Specification (Special Functions)

None

Additions to Chapter 6 of the GL Specification (State and State Requests)

None

Additions to the GLX Specification

None

GLX Protocol

None

Errors

INVALID_VALUE is generated by Material if enum is SHININESS and the shininess param is greater than the MAX_SHININESS_NV.

INVALID_VALUE is generated by Material if enum is SPOT_EXPONENT and the shininess param is greater than the MAX_SPOT_EXPONENT_NV.

New State

None.

New Implementation Dependent State

(table 6.24, p214) add the following entries:

Get Value	Type	Get Command	Minimum Value	Description	Sec	Attribute
MAX_SHININESS_NV	Z+	GetIntegerv	128	Maximum shininess for specular lighting	2.13.2	-
MAX_SPOT_EXPONENT_NV	Z+	GetIntegerv	128	Maximum exponent for spot lights	2.13.2	-

NVIDIA Implementation Details

NVIDIA's Release 4 drivers incorrectly and accidently advertised this extension with an "EXT" prefix instead of an "NV" prefix. Release 5 and later drivers correctly advertise this extension with an "NV" extension.

Revision History

5/20/00 - earlier versions of this specification had the incorrect enumerant values which did not match NVIDIA's driver implementation.

Name

    NV_multisample_filter_hint

Name Strings

    GL_NV_multisample_filter_hint

Notice

    Copyright NVIDIA Corporation, 2001.

IP Status

    NVIDIA Proprietary.

Status

    Shipping, May 2001

Version

    NVIDIA Date: May 16, 2001
    $Id: //sw/main/docs/OpenGL/specs/GL_NV_multisample_filter_hint.txt#2 $

Number

    259

Dependencies

    Written based on the OpenGL 1.2.1 specification.

    Requires ARB_multisample.

Overview

    OpenGL multisampling typically assumes that the samples of a given
    pixel are weighted uniformly and averaged to compute the pixel's
    resolved color. This extension provides a hint that permits
    implementations to provide an alternative method of resolving the
    color of multisampled pixels.

    As an example of such an alternative method, NVIDIA's GeForce3 GPU
    provides a technique known as Quincunx filtering. This technique
    is used in two-sample multisampling, but it blends the pixel's two
    samples and three additional samples from adjacent pixels. The sample
    pattern is analogous to the 5 pattern on a die. The quality of this
    technique is widely regarded as comparable to 4 sample multisampling.

Issues

    Is the glHint mechanism the right mechanism to expose this functionality?

      RESOLUTION: Yes. Multisample filtering quality is subject to
      the kinds of variations that the glHint was intended to control.

Arguably, the glHint mechanism only provides two definite settings: GL_FASTEST and GL_NICEST while there may be many different techniques for controlling multisample filtering quality. We expect hardware to support only one or two techniques rather than a multitude of nearly indistinguishable sampling techniques.

When does changing the multisampling filter hint take effect?

RESOLUTION: It may not be until the next swap buffers or glClear operation that the multisample hint actually takes effect. This may be implementation dependent.

What is the meaning of GL_DONT_CARE for the multisample hint?

RESOLUTION: By default, NVIDIA expects to treat GL_DONT_CARE the same as GL_FASTEST. However, the meaning of GL_DONT_CARE for this hint may be subject to a registry (or environment) setting, possibly settable through a control panel.

Does GL_NICEST require Quincunx filtering?

RESOLUTION: No. NVIDIA's GeForce3 Quincunx filtering is one possible technique that may be used to implement the GL_NICEST setting, but future GPUs may use other techniques.

Can the meaning of the multisample hint vary depending on the number of samples of the drawable?

RESOLUTION: Yes.

The following describes how GeForce3 uses the multisample hint:

When using 2-sample multisampling with GeForce3, the multisample filter hint affects multisample filtering as follows: GL_NICEST uses 5-tap Quincunx multisample filtering while GL_FASTEST uses standard even-weighted 2-tap multisample filtering of the pixel's 2 samples.

When using 4-sample multisampling with GeForce3, the multisample filter hint affects multisample filtering as follows: GL_NICEST uses 9-tap 3x3 multisample filtering while GL_FASTEST uses standard even-weighted 4-tap multisample filtering of the pixel's 4 samples.

What is the difference between a "tap" and a "sample"?

In the context of multisample filtering, a sample is a subpixel frame buffer sample containing color, depth, and stencil information. A tap is a source of data for filtering. Typically, samples are filtered by evenly weighting all the samples belonging to a pixel. In this case, the number of taps for the filter is equal to the number of samples for the pixel. In other filtering schemes, the number of taps and samples may not be equal (and potentially not evenly weighted as well). For example, GeForce3's quincunx filtering uses 5 taps even though each pixel has only 2 multisample samples. Three of the five taps source samples outside the pixel's footprint of two samples.

Should the multisample filtering technique be determined by the visual/PFD rather than OpenGL rendering context state?

RESOLUTION: No. The number of multisample samples per pixel that a window has is a property of the visual/PFD, but the filtering technique does not have to be defined up-front at when the pixel format is set.

While not quite consistent with the way ARB_multisample is specified, NVIDIA uses the SwapBuffers operation as a trigger for downsampling multisample sample buffers (other operations such as glReadPixels also trigger downsampling). But a SwapBuffers operation can be requested without a current OpenGL rendering context. What happens when a SwapBuffers operation is performed with no current OpenGL rendering context?

RESOLUTION: The multisample filter hint is treated as GL_DONT_CARE in this case. Applications that want the multisample filter hint to apply to their BufferSwap operation should perform the BufferSwap operation while bound to an OpenGL rendering context.

New Procedures and Functions

None

New Tokens

Accepted by the <target> parameter of Hint and by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

```
    MULTISAMPLE_FILTER_HINT_NV               0x8534
```

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

None

Additions to Chapter 3 of the GL Specification (Rasterization)

None

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the GL Specification (Special Functions)

-- Section 5.6 "Hints"

Replace the description of hint targets in the first paragraph with:

"target may be one of PERSPECTIVE_HINT, indicating the desired quality of parameter interpolation; POINT_SMOOTH_HINT, indicating the desired sampling quality of points; LINE_SMOOTH_HINT, indicating the desired sampling quality of lines; POLYGON_SMOOTH_HINT, indicating the desired sampling quality of polygons; FOG_HINT, indicating whether fog calculations are done per pixel or per vertex; and

     MULTISAMPLE_FILTER_HINT, indicating the desired quality of multisample
     filtering. The MULTISAMPLE_FILTER_HINT is ignored if the frame buffer
     has no multisample samples. When NICEST (or possibly DONT_CARE)
     multisample filtering is requested and the frame buffer supports
     multisampling, the multisample filter pattern may involve samples
     outside the pixel's sample set. The exact NICEST (or possibly
     DONT_CARE) multisample filtering technique used is implementation
     dependent and may vary with the number of multisample samples
     supported."

Additions to Chapter 6 of the GL Specification (State and State Requests)

None

Additions to the GLX, WGL, and AGL Specification

Add the following to the description of what happens at SwapBuffers time.

"When a SwapBuffers operation is performed by a thread without a current OpenGL rendering context and the target drawable to be swapped is multisampled, any multisample filtering operation that occurs should be done as if the GL_MULTISAMPLE_FILTER_HINT value is set to GL_DONT_CARE."

GLX Protocol

None

Errors

None

New State

(table 6.23, p213) add the following entry:

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
MULTISAMPLE_FILTER_HINT_NV	Z3	GetIntegerv	DONT_CARE	Multisample filter quality hint	5.6	hint

Revision History

None

Name

    NV_occlusion_query

Name Strings

    GL_NV_occlusion_query

Notice

    Copyright NVIDIA Corporation, 2001, 2002.

IP Status

    NVIDIA Proprietary.

Status

    Shipping (version 1.0)

Version

```
NVIDIA Date: February 6, 2002 (version 1.0)
$Id: //sw/main/docs/OpenGL/specs/GL_NV_occlusion_query.txt#3 $
```

Number

    261

Dependencies

    Written based on the wording of the OpenGL 1.3 specification.

    Requires support for the HP_occlusion_test extension.

Overview

```
The HP_occlusion_test extension defines a mechanism whereby an
application can query the visibility of an object, where "visible"
means that at least one pixel passes the depth and stencil tests.

The HP extension has two major shortcomings.

- It returns the result as a simple GL_TRUE/GL_FALSE result, when in
  fact it is often useful to know exactly how many pixels passed.
- It provides only a simple "stop-and-wait" model for using multiple
  queries.  The application begins an occlusion test and ends it;
  then, at some later point, it asks for the result, at which point
  the driver must stop and wait until the result from the previous
  test is back before the application can even begin the next one.
  This is a very simple model, but its performance is mediocre when
  an application wishes to perform many queries, and it eliminates
  most of the opportunites for parallelism between the CPU and GPU.

This extension solves both of those problems.  It returns as its
result the number of pixels that pass, and it provides an interface
conceptually similar to that of NV_fence that allows applications to
```

issue many occlusion queries before asking for the result of any one. As a result, they can overlap the time it takes for the occlusion query results to be returned with other, more useful work, such as rendering other parts of the scene or performing other computations on the CPU.

There are many situations where a pixel count, rather than a boolean result, is useful.

- If the visibility test is an object bounding box being used to decide whether to skip the object, sometimes it can be acceptable, and beneficial to performance, to skip an object if less than some threshold number of pixels could be visible.
- Knowing the number of pixels visible in the bounding box may also help decide what level of detail a model should be drawn with. If only a few pixels are visible, a low-detail model may be acceptable. In general, this allows level-of-detail mechanisms to be slightly less ad hoc.
- "Depth peeling" techniques, such as order-independent transparency, would typically like to know when to stop rendering more layers; it is difficult to come up with a way to determine a priori how many layers to use. A boolean count allows applications to stop when more layers will not affect the image at all, but this will likely be unacceptable for performance, with minimal gains to image quality. Instead, it makes more sense to stop rendering when the number of pixels goes below a threshold; this should provide better results than any of these other algorithms.
- Occlusion queries can be used as a replacement for glReadPixels of the depth buffer to determine whether, say, a light source is visible for the purposes of a lens flare effect or a halo to simulate glare. Pixel counts allow you to compute the percentage of the light source that is visible, and the brightness of these effects can be modulated accordingly.

Issues

* *Should we use an object-based interface?*

RESOLVED: Yes, this makes the interface much simpler, and it is friendly for indirect rendering.

* *Should we offer an entry point analogous to glTestFenceNV?*

RESOLVED: No, it is sufficient to have glGetOcclusionQueryivNV provide a query for whether the occlusion query result is back yet. Whereas it is interesting to poll fence objects, it is relatively less interesting to poll occlusion queries.

* *Is glGetOcclusionQueryuivNV necessary?*

RESOLVED: Yes, it makes using a 32-bit pixel count less painful.

* *Should there be a limit on how many queries can be outstanding?*

RESOLVED: No. This would make the extension much more difficult to spec and use. Allowing this does not add any significant implementation burden; and even if drivers have some internal limit on the number of outstanding queries, it is not expected that applications will need to know this to achieve optimal or near-optimal performance.

* *What happens if glBeginOcclusionQueryNV is called when an occlusion query is already outstanding for a different object?*

RESOLVED: This is a GL_INVALID_OPERATION error.

* *What happens if HP_occlusion_test and NV_occlusion_query usage is overlapped?*

RESOLVED: The two can be overlapped safely. Counting is enabled if we are _either_ inside a glBeginOcclusionQueryNV or if if GL_OCCLUSION_TEST_HP is enabled. The alternative (producing an error) does not work -- it would require that glPopAttrib be capable of producing an error, which would be rather problematic.

Note that glBeginOcclusionQueryNV, not glEndOcclusionQueryNV, resets the pixel counter and occlusion test result. This can avoid certain types of strange behavior where an occlusion query's pixel count does not always correspond to the pixels rendered during the occlusion query. The spec would make sense the other way, but the behavior would be strange.

* *Does EndOcclusionQuery need to take any parameters?*

RESOLVED: No. Giving it, for example, an "id" parameter would be redundant -- adding complexity for no benefit. Only one query can be active at a time.

* *How many bits should we require the pixel counter to be, at minimum?*

RESOLVED: 24. 24 is enough to handle 8.7 full overdraws of a 1600x1200 window. That seems quite sufficient.

* *What should we do about overflows?*

RESOLVED: Overflows leave the pixel count undefined. Saturating is recommended but not required.

The ideal behavior really is to saturate. This ensures that you always get a "large" result when you render many pixels. It also ensures that apps which want a boolean test can do one on their own, and not worry about the rare case where the result ends up exactly at zero from wrapping.

That being said, with 24 bits of pixel count required, it's not clear that this really matters. It's better to be a bit permissive here. In addition, even if saturation was required, the goal of having strictly defined behavior is still not really met.

Applications don't (or at least shouldn't) check for some _exact_ number of bits. Imagine if a multitextured app had been written that required that the number of texture units supported be _exactly_ two! Implementors of OpenGL would be greatly annoyed to find that the app did not run on, say, three-texture or four-texture hardware.

So, we expect apps here to always be doing a "greater than or equal to" check. An app might check for, say, at least 28 bits. This doesn't ensure defined behavior -- it only ensures that once an overflow occurs (which may happen at any power of two), that overflow will be handled with saturation. This behavior still remains sufficiently unpredictable that the reasons for defining behavior in even rarely-used cases (preventing compatibility problems, for example) are unsatisfied.

All that having been said, saturation is still explicitly recommended in the spec language.

*   *What is the interaction with multisample, which was not defined in the original spec?*

RESOLVED: The pixel count is the number of samples that pass, not the number of pixels. This is true even if GL_MULTISAMPLE is disabled but GL_SAMPLE_BUFFERS is 1. Note that the depth/stencil test optimization whereby implementations may choose to depth test at only one of the samples when GL_MULTISAMPLE is disabled does not cause this to become ill-specified, because we are counting the number of samples that are still alive _after_ the depth test stage. The mechanism used to decide whether to kill or keep those samples is not relevant.

*   *Exactly what stage are we counting at? The original spec said depth test; what does stencil test do?*

RESOLVED: We are counting immediately after _both_ the depth and stencil tests, i.e., pixels that pass both. This was the original spec's intent. Note that the depth test comes after the stencil test, so to say that it is the number that pass the depth test is reasonable; though it is often helpful to think of the depth and stencil tests as being combined, because the depth test result impacts the stencil operation used.

*   *Is it guaranteed that occlusion queries return in order?*

RESOLVED: Yes. It makes sense to do this. If occlusion test X occurred before occlusion query Y, and the driver informs the app that occlusion query Y is done, the app can infer that occlusion query X is also done. For applications that do poll, this allows them to do so with less effort.

*   *Will polling an occlusion query without a glFlush possibly cause an infinite loop?*

RESOLVED: Yes, this is a risk. If you ask for the result, however, any flush required will be done automatically. It is only when you are polling that this is a problem because there is no guarantee that a flush has occured in the time since glEndOcclusionQueryNV, and the spec is written to say that the result is only "available" if the value could be returned

```
    _instantaneously_.

    This is different from NV_fence, where FinishFenceNV can cause an
    app hang, and where TestFenceNV was also not guaranteed to ever
    finish.

    There need not be any spec language to describe this behavior
    because it is implied by what is already said.

    In short, if you use GL_PIXEL_COUNT_AVAILABLE_NV, you _must_ use
    glFlush, or your app may hang.

*   The HP_occlusion_test specs did not contain the spec edits that
    explain the exact way the extension works.  Should this spec fill
    in those details?

    RESOLVED: Yes.  These two extensions are intertwined in so many
    important ways that doing so is not optional.

*   Should there be a "target" parameter to BeginOcclusionQuery?

    RESOLVED: No.  We're not trying to solve the problem of "query
    anything" here.

*   What might an application that uses this extension look like?

    Here is some rough sample code:

    GLuint occlusionQueries[N];
    GLuint pixelCount;

    glGenOcclusionQueriesNV(N, occlusionQueries);
    ...
    // before this point, render major occluders
    glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
    glDepthMask(GL_FALSE);
    // also disable texturing and any fancy shading features
    for (i = 0; i < N; i++) {
        glBeginOcclusionQueryNV(occlusionQueries[i]);
        // render bounding box for object i
        glEndOcclusionQueryNV();
    }
    // at this point, if possible, go and do some other computation
    glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
    glDepthMask(GL_TRUE);
    // reenable other state
    for (i = 0; i < N; i++) {
        glGetOcclusionQueryuivNV(occlusionQueries[i], GL_PIXEL_COUNT_NV,
                                 &pixelCount);
        if (pixelCount > 0) {
            // render object i
        }
    }

*   Is this extension useful for saving geometry, fill rate, or both?

    It is expected that it will be most useful for saving geometry
```

work, because for the cost of rendering a bounding box you can save rendering a normal object.

It is possible for this extension to help in fill-limited situations, but using it may also hurt performance in such situations, because rendering the pixels of a bounding box is hardly free. In most situations a bounding box will probably have more pixels than the original object.

One exception is that for objects rendered with multiple passes, the first pass can be wrapped with an occlusion query almost for free. That is, render the first pass for all objects in the scene, and get the number of pixels rendered on each object. If zero pixels were rendered for an object, you can skip subsequent rendering passes. This trick can be very useful in many cases.

*   *What can be said about guaranteeing correctness when using occlusion queries, especially as it relates to invariance?*

Invariance is critical to guarantee the correctness of occlusion queries. If occlusion queries go through a different code path than standard rendering, the pixels rendered may be different.

However, the invariance issues are difficult at best to solve. Because of the vagaries of floating-point precision, it is difficult to guarantee that rendering a bounding box will render at least as many pixels with equal or smaller Z values than the object itself would have rendered.

Likewise, many other aspects of rendering state tend to be different when performing an occlusion query. Color and depth writes are typically disabled, as are texturing, vertex programs, and any fancy per-pixel math. So unless all these features have guarantees of invariance themselves (unlikely at best), requiring invariance for NV_occlusion_query would be futile.

For what it's worth, NVIDIA's implementation is fully invariant with respect to whether an occlusion query is active; that is, it does not affect the operation of any other stage of the pipeline. (When occlusion queries are being emulated on hardware that does not support them, via the emulation registry keys, using an occlusion query produces a software rasteriation fallback, and in such cases invariance cannot be guaranteed.)

Another problem that can threaten correctness is near and far clipping. If the bounding box penetrates the near clip plane, for example, it may be clipped away, reducing the number of pixels counted, when in fact the original object may have stayed entirely beyond the near clip plane. Whenever you design an algorithm using occlusion queries, it is best to be careful about the near and far clip planes.

*   *How can frame-to-frame coherency help applications using this extension get even higher performance?*

Usually, if an object is visible one frame, it will be visible the next frame, and if it is not visible, it will not be visible the next frame.

Of course, for most applications, "usually" isn't good enough. It is undesirable, but acceptable, to render an object that wasn't visible, because that only costs performance. It is generally unacceptable to not render an object that was visible.

The simplest approach is that visible objects should be checked every N frames (where, say, N=5) to see if they have become occluded, while objects that were occluded last frame must be rechecked again in the current frame to guarantee that they are still occluded. This will reduce the number of wasteful occlusion queries by a factor of almost N.

It may also pay to do a raycast on the CPU in order to try to prove that an object is visible. After all, occlusion queries are only one of many items in your bag of tricks to decide whether objects are visible or invisible. They are not an excuse to skip frustum culling, or precomputing visibility using portals for static environments, or other standard visibility techniques.

In general, though, taking advantage of frame-to-frame coherency in your occlusion query code is absolutely essential to getting the best possible performance.

New Procedures and Functions

```
void GenOcclusionQueriesNV(sizei n, uint *ids);
void DeleteOcclusionQueriesNV(sizei n, const uint *ids);
boolean IsOcclusionQueryNV(uint id);
void BeginOcclusionQueryNV(uint id);
void EndOcclusionQueryNV(void);
void GetOcclusionQueryivNV(uint id, enum pname, int *params);
void GetOcclusionQueryuivNV(uint id, enum pname, uint *params);
```

New Tokens

Accepted by the <cap> parameter of Enable, Disable, and IsEnabled, and by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

```
    OCCLUSION_TEST_HP                              0x8165
```

Accepted by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

```
    OCCLUSION_TEST_RESULT_HP                       0x8166
    PIXEL_COUNTER_BITS_NV                          0x8864
    CURRENT_OCCLUSION_QUERY_ID_NV                  0x8865
Accepted by the <pname> parameter of GetOcclusionQueryivNV and
GetOcclusionQueryuivNV:

    PIXEL_COUNT_NV                                  0x8866
    PIXEL_COUNT_AVAILABLE_NV                        0x8867
```

Additions to Chapter 2 of the OpenGL 1.3 Specification (OpenGL Operation)

None.

Additions to Chapter 3 of the OpenGL 1.3 Specification (Rasterization)

None.

Additions to Chapter 4 of the OpenGL 1.3 Specification (Per-Fragment Operations and the Frame Buffer)

Add a new section "Occlusion Tests and Queries" between sections 4.1.6 and 4.1.7:

"4.1.6A Occlusion Tests and Queries

Occlusion testing keeps track of whether any pixels have passed the depth test. Such testing is enabled or disabled with the generic Enable and Disable commands using the symbolic constant OCCLUSION_TEST_HP. The occlusion test result is initially FALSE.

Occlusion queries can be used to track the exact number of fragments that pass the depth test. Occlusion queries are associated with occlusion query objects. The command

```
  void GenOcclusionQueriesNV(sizei n, uint *ids);
```

returns n previously unused occlusion query names in ids. These names are marked as used, but no object is associated with them until the first time BeginOcclusionQueryNV is called on them. Occlusion queries contain one piece of state, a pixel count result. This pixel count result is initialized to zero when the object is created.

Occlusion queries are deleted by calling

```
  void DeleteOcclusionQueriesNV(sizei n, const uint *ids);
```

ids contains n names of occlusion queries to be deleted. After an occlusion query is deleted, its name is again unused. Unused names in ids are silently ignored.

An occlusion query can be started and finished by calling

```
  void BeginOcclusionQueryNV(uint id);
  void EndOcclusionQueryNV(void);
```

If BeginOcclusionQueryNV is called with an unused id, that id is marked as used and associated with a new occlusion query object. If it is called while another occlusion query is active, an INVALID_OPERATION error is generated. If EndOcclusionQueryNV is called while no occlusion query is active, an INVALID_OPERATION error is generated. Calling either GenOCclusionQueriesNV or DeleteOcclusionQueriesNV while an occlusion query is active causes an INVALID_OPERATION error to be generated.

When EndOcclusionQueryNV is called, the current pixel counter is copied into the active occlusion query object's pixel count result. BeginOcclusionQueryNV resets the pixel counter to zero and the occlusion test result to FALSE.

Whenever a fragment reaches this stage and OCCLUSION_TEST_HP is enabled or an occlusion query is active, the occlusion test result is set to TRUE and the pixel counter is incremented. If the value of SAMPLE_BUFFERS is 1, then the pixel counter is incremented by the number of samples whose coverage bit is set; otherwise, it is always incremented by one. If it the pixel counter overflows, i.e., exceeds the value 2^PIXEL_COUNTER_BITS_NV-1, its value becomes undefined. It is recommended, but not required, that implementations handle this overflow case by saturating at 2^PIXEL_COUNTER_BITS_NV-1 and incrementing no further.

The necessary state is a single bit indicating whether the occlusion test is enabled, a single bit indicating whether an occlusion query is active, the identifier of the currently active occlusion query, a counter of no smaller than 24 bits keeping track of the pixel count, and a single bit indicating the occlusion test result."

Additions to Chapter 5 of the OpenGL 1.3 Specification (Special Functions)

Add to the end of Section 5.4 "Display Lists":

"DeleteOcclusionQueriesNV, GenOcclusionQueriesNV, IsOcclusionQueryNV, GetOcclusionQueryivNV, and GetOcclusionQueryuivNV are not complied into display lists but are executed immediately."

Additions to Chapter 6 of the OpenGL 1.3 Specification (State and State Requests)

Add a new section 6.1.13 "Occlusion Test and Occlusion Queries":

"The occlusion test result can be queried using GetBooleanv, GetIntegerv, GetFloatv, or GetDoublev with a <pname> of OCCLUSION_TEST_RESULT_HP. Whenever such a query is performed, the occlusion test result is reset to FALSE and the pixel counter is reset to zero as a side effect.

Which occlusion query is active can be queried using GetBooleanv, GetIntegerv, GetFloatv, or GetDoublev with a <pname> of CURRENT_OCCLUSION_QUERY_ID_NV. This query returns the name of the currently active occlusion query if one is active, and zero otherwise.

The state of an occlusion query can be queried with the commands

```
  void GetOcclusionQueryivNV(uint id, enum pname, int *params);
  void GetOcclusionQueryuivNV(uint id, enum pname, uint *params);
```

```
    If the occlusion query object named by id is currently active, then
    an INVALID_OPERATION error is generated.

    If <pname> is PIXEL_COUNT_NV, then the occlusion query's pixel count
    result is placed in params.

    Often, occlusion query results will be returned asychronously with
    respect to the host processor's operation.  As a result, sometimes,
    if a pixel count is queried, the host must wait until the result is
    back.  If <pname> is PIXEL_COUNT_AVAILABLE_NV, the value placed in
    params indicates whether or not such a wait would occur if the pixel
    count for that occlusion query were to be queried presently.  A
    result of TRUE means no wait would be required; a result of FALSE
    means that some wait would occur.  The length of this wait is
    potentially unbounded.  It must always be true that if the result for
    one occlusion query is available, the result for all previous
    occlusion queries must also be available at that point in time."
```

GLX Protocol

```
    Seven new GL commands are added.

    The following two rendering commands are sent to the server as part
    of a glXRender request:

        BeginOcclusionQueryNV
            2           8               rendering command length
            2           ????            rendering command opcode
            4           CARD32          id

        EndOcclusionQueryNV
            2           4               rendering command length
            2           ????            rendering command opcode

    The remaining fivecommands are non-rendering commands.  These
    commands are sent separately (i.e., not as part of a glXRender or
    glXRenderLarge request), using the glXVendorPrivateWithReply
    request:

        DeleteOcclusionQueriesNV
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           4+n             request length
            4           ????            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           INT32           n
            n*4         LISTofCARD32    ids
        GenOcclusionQueriesNV
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           4               request length
            4           ????            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           INT32           n
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           n               reply length
            24                          unused
            n*4         LISTofCARD322   queries

        IsOcclusionQueryNV
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           4               request length
            4           ????            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           CARD32          id
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           0               reply length
            4           BOOL32          return value
            20                          unused
            1           1               reply

        GetOcclusionQueryivNV
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           5               request length
            4           ????            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           CARD32          id
            4           ENUM            pname
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           m               reply length, m=(n==1?0:n)
            4                           unused
            4           CARD32          n

            if (n=1) this follows:

            4           INT32           params
            12                          unused

            otherwise this follows:

            16                          unused
            n*4         LISTofINT32     params
        GetOcclusionQueryuivNV
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           5               request length
            4           ????            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           CARD32          id
            4           ENUM            pname
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           m               reply length, m=(n==1?0:n)
            4                           unused
            4           CARD32          n

            if (n=1) this follows:

            4           CARD32          params
            12                          unused

            otherwise this follows:

            16                          unused
            n*4         LISTofCARD32    params
```

Errors

The error INVALID_VALUE is generated if GenOcclusionQueriesNV is called where n is negative.

The error INVALID_VALUE is generated if DeleteOcclusionQueriesNV is called where n is negative.

The error INVALID_OPERATION is generated if GenOcclusionQueriesNV or DeleteOcclusionQueriesNV is called when an occlusion query is active.

The error INVALID_OPERATION is generated if BeginOcclusionQueryNV is called when an occlusion query is already active.

The error INVALID_OPERATION is generated if EndOcclusionQueryNV is called when an occlusion query is not active.

The error INVALID_OPERATION is generated if GetOcclusionQueryivNV or GetOcclusionQueryuivNV is called where id is not the name of an occlusion query.

The error INVALID_OPERATION is generated if GetOcclusionQueryivNV or GetOcclusionQueryuivNV is called where id is the name of the currently active occlusion query.

The error INVALID_ENUM is generated if GetOcclusionQueryivNV or GetOcclusionQueryuivNV is called where pname is not either PIXEL_COUNT_NV or PIXEL_COUNT_AVAILABLE_NV.

    The error INVALID_OPERATION is generated if any of the commands
    defined in this extension is executed between the execution of Begin
    and the corresponding execution of End.

New State

(table 6.18, p. 226)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
OCCLUSION_TEST_HP	B	IsEnabled	FALSE	occlusion test enable	4.1.6A	enable
OCCLUSION_TEST_RESULT_HP	B	GetBooleanv	FALSE	occlusion test result	4.1.6A	-
-	B	GetBooleanv	FALSE	occlusion query active	4.1.6A	-
CURRENT_OCCLUSION_QUERY_ID_NV	Z+	GetIntegerv	0	occlusion query ID	4.1.6A	-
-	Z+	-	0	pixel counter	4.1.6A	-

New Implementation Dependent State

(table 6.29, p. 237) Add the following entry:

Get Value	Type	Get Command	Minimum Value	Description	Sec	Attribute
PIXEL_COUNTER_BITS_NV	Z+	GetIntegerv	24	Number of bits in pixel counters	6.1.13	-

Revision History

    none yet

Name

    NV_packed_depth_stencil

Name Strings

    GL_NV_packed_depth_stencil

Notice

IP Status

    NVIDIA Proprietary.

Version

```
    NVIDIA Date: January 18, 2001
    $Id: //sw/main/docs/OpenGL/specs/GL_NV_packed_depth_stencil.txt#6 $
```

Number

    ??

Dependencies

Written based on the wording of the OpenGL 1.2.1 specification.

SGIX_depth_texture affects the definition of this extension.

Overview

Many OpenGL implementations have chosen to interleave the depth and stencil buffers into one buffer, often with 24 bits of depth precision and 8 bits of stencil data. 32 bits is more than is needed for the depth buffer much of the time; a 24-bit depth buffer, on the other hand, requires that reads and writes of depth data be unaligned with respect to power-of-two boundaries. On the other hand, 8 bits of stencil data is more than sufficient for most applications, so it is only natural to pack the two buffers into a single buffer with both depth and stencil data. OpenGL never provides direct access to the buffers, so the OpenGL implementation can provide an interface to applications where it appears the one merged buffer is composed of two logical buffers.

One disadvantage of this scheme is that OpenGL lacks any means by which this packed data can be handled efficiently. For example, when an application reads from the 24-bit depth buffer, using the type GL_UNSIGNED_SHORT will lose 8 bits of data, while GL_UNSIGNED_INT has 8 too many. Both require expensive format conversion operations. A 24-bit format would be no more suitable, because it would also suffer from the unaligned memory accesses that made the standalone 24-bit depth buffer an unattractive proposition in the first place.

Many applications, such as parallel rendering applications, may also wish to draw to or read back from both the depth and stencil buffers at the same time. Currently this requires two separate operations, reducing performance. Since the buffers are interleaved, drawing to or reading from both should be no more expensive than using just one; in some cases, it may even be cheaper.

This extension provides a new data format, GL_DEPTH_STENCIL_NV, that can be used with the glDrawPixels, glReadPixels, and glCopyPixels commands, as well as a packed data type, GL_UNSIGNED_INT_24_8_NV, that is meant to be used with GL_DEPTH_STENCIL_NV. No other formats are supported with GL_DEPTH_STENCIL_NV. If SGIX_depth_texture is supported, GL_DEPTH_STENCIL_NV/GL_UNSIGNED_INT_24_8_NV data can also be used for textures; this provides a more efficient way to supply data for a 24-bit depth texture.

GL_DEPTH_STENCIL_NV data, when passed through the pixel path, undergoes both depth and stencil operations. The depth data is scaled and biased by the current GL_DEPTH_SCALE and GL_DEPTH_BIAS, while the stencil data is shifted and offset by the current GL_INDEX_SHIFT and GL_INDEX_OFFSET. The stencil data is also put through the stencil-to-stencil pixel map.

glDrawPixels of GL_DEPTH_STENCIL_NV data operates similarly to that of GL_STENCIL_INDEX data, bypassing the OpenGL fragment pipeline entirely, unlike the treatment of GL_DEPTH_COMPONENT data. The stencil and depth masks are applied, as are the pixel ownership and scissor tests, but all other operations are skipped.

glReadPixels of GL_DEPTH_STENCIL_NV data reads back a rectangle from both the depth and stencil buffers.

glCopyPixels of GL_DEPTH_STENCIL_NV data copies a rectangle from both the depth and stencil buffers. Like glDrawPixels, it applies both the stencil and depth masks but skips the remainder of the OpenGL fragment pipeline.

glTex[Sub]Image[1,2,3]D of GL_DEPTH_STENCIL_NV data loads depth data into a depth texture. glGetTexImage of GL_DEPTH_STENCIL_NV data can be used to retrieve depth data from a depth texture.

Issues

* Depth data has a format of GL_DEPTH_COMPONENT, and stencil data has a format of GL_STENCIL_INDEX. So shouldn't the enumerant be called GL_DEPTH_COMPONENT_STENCIL_INDEX_NV?

  RESOLVED: No, this is fairly clumsy.

* Should we support CopyPixels?

  RESOLVED: Yes. Right now copying stencil data means masking off just the stencil bits, while copying depth data has strange unintended consequences (fragment operations) and is difficult to implement. It is easy and useful to add CopyPixels support.

```
    *   Should we support textures?

        RESOLVED: Yes.  24-bit depth textures have no good format without
        this extension.

    *   Should the depth/stencil format support other standard types,
        like GL_FLOAT or GL_UNSIGNED_INT?

        RESOLVED: No, this extension is designed to be minimalist.
        Supporting more types gains little because the new types will
        just require data format conversions.  Our goal is to match the
        native format of the buffer, not add broad new classes of
        functionality.

    *   Should the 24/8 format be supported for other formats, such as
        LUMINANCE_ALPHA?  Should we support an 8/24 reversed format as
        well?

        RESOLVED: No and no, this adds implementation burden and gains us
        little, if anything.

    *   Does anything need to be written in the spec on the topic of
        using GL_DEPTH_STENCIL_NV formats for glTexImage* or
        glGetTexImage?

        RESOLVED: No.  Since the SGIX_depth_texture extension spec was
        never actually written (the additions to Section 3 are "XXX -
        lots" and a few brief notes on how it's intended to work), it's
        impossible to write what would essentially be amendments to that
        spec.

        However, it is worthwhile to mention here the intended behavior.
        When downloading into a depth component texture, the stencil
        indices are ignored, and when retrieving a depth component
        texture, the stencil indices obtained from the texture are
        undefined.

    *   Should anything be said about performance?

        RESOLVED: No, not in the spec.  However, common sense should
        apply.  Apps should probably check that GL_DEPTH_BITS is 24 and
        that GL_STENCIL_BITS is 8 before using either the new DrawPixels
        or ReadPixels formats.  CopyPixels is probably safe regardless of
        the size of either buffer.  The 24/8 format should probably only
        be used with 24-bit depth textures.
```

New Procedures and Functions

```
    None.
```

New Tokens

    Accepted by the <format> parameter of DrawPixels, ReadPixels,
    TexImage1D, TexImage2D, TexImage3D, TexSubImage1D, TexSubImage2D,
    TexSubImage3D, and GetTexImage, and by the <type> parameter of
    CopyPixels:

        DEPTH_STENCIL_NV   0x84F9

    Accepted by the <type> parameter of DrawPixels, ReadPixels,
    TexImage1D, TexImage2D, TexImage3D, TexSubImage1D, TexSubImage2D,
    TexSubImage3D, and GetTexImage:

        UNSIGNED_INT_24_8_NV   0x84FA

Additions to Chapter 2 of the OpenGL 1.2.1 Specification (OpenGL Operation)

    None.

Additions to Chapter 3 of the OpenGL 1.2.1 Specification (Rasterization)

    Update the first paragraph on page 90 to say:

    "... If the GL is in color index mode and <format> is not one of
    COLOR_INDEX, STENCIL_INDEX, DEPTH_COMPONENT, or DEPTH_STENCIL_NV,
    then the error INVALID_OPERATION occurs. If <type> is BITMAP and
    <format> is not COLOR_INDEX or STENCIL_INDEX then the error
    INVALID_ENUM occurs. If <format> is DEPTH_STENCIL_NV and <type> is
    not UNSIGNED_INT_24_8_NV then the error INVALID_ENUM occurs. Some
    additional constraints on the combinations of <format> and <type>
    values that are accepted is discussed below."

    Add a row to Table 3.5 (page 91):

type Parameter	GL Type	Special
...	...	...
UNSIGNED_INT_2_10_10_10_REV	uint	Yes
UNSIGNED_INT_24_8_NV	uint	Yes

    Add a row to Table 3.6 (page 92):

Format Name	Element Meaning and Order	Target Buffer
...	...	...
DEPTH_COMPONENT	Depth	Depth
DEPTH_STENCIL_NV	Depth and Stencil Index	Depth and Stencil
...	...	...

    Add a row to Table 3.8 (page 94):

type Parameter	GL Type	Components	Pixel Formats
...	...	...	...
UNSIGNED_INT_2_10_10_10_REV	uint	4	RGBA,BGRA
UNSIGNED_INT_24_8_NV	uint	2	DEPTH_STENCIL_NV

```
    Update the last paragraph on page 93 to say:

    "Calling DrawPixels with a <type> of UNSIGNED_BYTE_3_3_2, ...,
    UNSIGNED_INT_2_10_10_10_REV, or UNSIGNED_INT_24_8_NV is a special
    case in which all the components of each group are packed into a
    single unsigned byte, unsigned short, or unsigned int, depending on
    the type."

    Add the following diagram to Table 3.11 (page 97):

    UNSIGNED_INT_24_8_NV

       31 30 29 28 27 26 ... 12 11 10 9 8 7 6 5 4 3 2 1 0
      +----------------------------------+---------------+
      |           1st Component          | 2nd Component |
      +----------------------------------+---------------+

    Add a row to Table 3.12 (page 98):

      Format            |  1st     2nd     3rd     4th
      ------------------+------------------------------
      ...               |  ...     ...     ...     ...
      BGRA              |  blue    green   red     alpha
      DEPTH_STENCIL_NV  |  depth   stencil N/A     N/A

    Add the following paragraph to the end of the section "Conversion to
    floating-point" (page 99):

    "For groups of components that contain both standard components and
    index elements, such as DEPTH_STENCIL_NV, the index elements are not
    converted."

    Update the last paragraph in the section "Conversion to Fragments"
    (page 100) to say:

    "... Groups arising from DrawPixels with a <format> of STENCIL_INDEX
    or DEPTH_STENCIL_NV are treated specially and are described in
    section 4.3.1."

    Update the first paragraph of section 3.6.5 "Pixel Transfer
    Operations" (pages 100-101) to say:

    "The GL defines five kinds of pixel groups:

       1. RGBA component: Each group comprises four color components:
          red, green, blue, and alpha.
       2. Depth component: Each group comprises a single depth component.
       3. Color index: Each group comprises a single color index.
       4. Stencil index: Each group comprises a single stencil index.
       5. Depth/stencil: Each group comprises a depth component and a
          stencil index."

    Update the first paragraph in the section "Arithmetic on Components"
    (page 101) to say:

    "This step applies only to RGBA component and depth component groups
    and the depth components in depth/stencil groups. ..."
```

Update the first paragraph in the section "Arithmetic on Indices" (page 101) to say:

"This step applies only to color index and stencil index groups and the stencil indices in depth/stencil groups. ..."

Update the first paragraph in the section "Stencil Index Lookup" (page 102) to say:

"This step applies only to stencil index groups and the stencil indices in depth/stencil groups. ..."

Additions to Chapter 4 of the OpenGL 1.2.1 Specification (Per-Fragment Operations and the Frame Buffer)

Replace section 4.3.1 "Writing to the Stencil Buffer" (page 156) with the following:

"4.3.1 Writing to the Stencil Buffer or to the Depth and Stencil Buffers

The operation of DrawPixels was described in section 3.6.4, except if the <format> argument was STENCIL_INDEX or DEPTH_STENCIL_NV. In this case, all operations described for DrawPixels take place, but window (x,y) coordinates, each with the corresponding stencil index or depth value and stencil index, are produced in lieu of fragments. Each coordinate-data pair is sent directly to the per-fragment operations, bypassing the texture, fog, and antialiasing application stages of rasterization. Each pair is then treated as a fragment for purposes of the pixel ownership and scissor tests; all other per-fragment operations are bypassed. Finally, each stencil index is written to its indicated location in the framebuffer, subject to the current setting of StencilMask, and if a depth component is present, if the setting of DepthMask is not FALSE, it is also written to the framebuffer; the setting of DepthTest is ignored.

The error INVALID_OPERATION results if there is no stencil buffer, or if the <format> argument was DEPTH_STENCIL_NV, if there is no depth buffer."

Add the following paragraph after the second paragraph of the section "Obtaining Pixels from the Framebuffer" (page 158):

"If the <format> is DEPTH_STENCIL_NV, then values are taken from both the depth buffer and the stencil buffer. If there is no depth buffer or if there is no stencil buffer, the error INVALID_OPERATION occurs. If the <type> parameter is not UNSIGNED_INT_24_8_NV, the error INVALID_ENUM occurs."

Update the third paragraph on page 159 to say:

"If the GL is in RGBA mode, and <format> is one of RED, GREEN, BLUE, ALPHA, RGB, RGBA, BGR, BGRA, LUMINANCE, or LUMINANCE_ALPHA, then red, green, blue, and alpha values are obtained from the framebuffer

Update the first sentence of the section "Conversion of RGBA values" (page 159) to say:

"This step applies only if the GL is in RGBA mode, and then only if <format> is neither STENCIL_INDEX, DEPTH_COMPONENT, nor DEPTH_STENCIL_NV."

Update the section "Conversion of Depth values" (page 159) to say:

"This step applies only if <format> is DEPTH_COMPONENT or DEPTH_STENCIL_NV. Each element taken from the depth buffer is taken to be a fixed-point value in [0,1] with m bits, where m is the number of bits in the depth buffer (see section 2.10.1)."

Add a row to Table 4.6 (page 160):

type Parameter	Index Mask
...	...
INT	2^31-1
UNSIGNED_INT_24_8_NV	2^8-1

Add the following paragraph to the end of the section "Final Conversion" (page 160):

"For a depth/stencil pair, first the depth component is clamped to [0,1]. Then the appropriate conversion formula from Table 4.7 is applied to the depth component, while the index is masked by the value given in Table 4.6 or converted to a GL float data type if the <type> is FLOAT."

Add a row to Table 4.7 (page 161):

type Parameter	GL Type	Component Conversion ...
...	...	...
UNSIGNED_INT_2_10_10_10_REV	uint	c = (2^N - 1)f
UNSIGNED_INT_24_8_NV	uint	c = (2^N - 1)f (depth only)

Update the second and third paragraphs of section 4.3.3 (page 162) to say:

"<type> is a symbolic constant that must be one of COLOR, STENCIL, DEPTH, or DEPTH_STENCIL_NV, indicating that the values to be transfered are colors, stencil values, or depth values, respectively. The first four arguments have the same interpretation as the corresponding arguments to ReadPixels.

Values are obtained from the framebuffer, converted (if appropriate), then subjected to the pixel transfer operations described in section 3.6.5, just as if ReadPixels were called with the corresponding arguments. If the <type> is STENCIL, DEPTH, or DEPTH_STENCIL_NV, then it is as if the <format> for ReadPixels were STENCIL_INDEX, DEPTH_COMPONENT, or DEPTH_STENCIL_NV, respectively. If the <type> is COLOR, then if the GL is in RGBA mode, it is as if the <format> were RGBA, while if the GL is in color index mode, it is as if the <format> were COLOR_INDEX."

Additions to Chapter 5 of the OpenGL 1.2.1 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 1.2.1 Specification (State and State Requests)

None.

GLX Protocol

None.

Errors

The error INVALID_ENUM is generated if DrawPixels or ReadPixels is called where format is DEPTH_STENCIL_NV and type is not UNSIGNED_INT_24_8_NV.

The error INVALID_OPERATION is generated if DrawPixels or ReadPixels is called where type is UNSIGNED_INT_24_8_NV and format is not DEPTH_STENCIL_NV.

The error INVALID_OPERATION is generated if DrawPixels or ReadPixels is called where format is DEPTH_STENCIL_NV and there is not both a depth buffer and a stencil buffer.

The error INVALID_OPERATION is generated if CopyPixels is called where type is DEPTH_STENCIL_NV and there is not both a depth buffer and a stencil buffer.

New State

None.

Revision History

None yet

Name

    NV_parameter_buffer_object

Name Strings

    None (impled by NV_GPU_program4)

Contact

    Pat Brown, NVIDIA Corporation (pbrown 'at' nvidia.com)
    Eric Werness, NVIDIA Corporation (ewerness 'at' nvidia.com)

Status

    Shipping for GeForce 8 Series (November 2006)

Version

```
    Last Modified Date:         04/18/2007
    NVIDIA Revision:            7
```

Number

    339

Dependencies

    OpenGL 2.0 is required.

    NV_gpu_program4 is required.

    This extension is written against the OpenGL 2.0 specification.

    NV_transform_feedback affects this extension.

Overview

This extension, in conjunction with NV_gpu_program4, provides a new type of program parameter than can be used as a constant during vertex, fragment, or geometry program execution. Each program target has a set of parameter buffer binding points to which buffer objects can be attached.

A vertex, fragment, or geometry program can read data from the attached buffer objects using a binding of the form "program.buffer[a][b]". This binding reads data from the buffer object attached to binding point <a>. The buffer object attached is treated either as an array of 32-bit words or an array of four-component vectors, and the binding above reads the array element numbered <b>.

The use of buffer objects allows applications to change large blocks of program parameters at once, simply by binding a new buffer object. It also provides a number of new ways to load parameter values, including readback from the frame buffer (EXT_pixel_buffer_object), transform feedback (NV_transform_feedback), buffer object loading functions such as MapBuffer and BufferData, as well as dedicated parameter buffer update functions provided by this extension.

New Procedures and Functions

```
    void BindBufferRangeNV(enum target, uint index, uint buffer,
                           intptr offset, sizeiptr size);
    void BindBufferOffsetNV(enum target, uint index, uint buffer,
                            intptr offset);
    void BindBufferBaseNV(enum target, uint index, uint buffer);
    void ProgramBufferParametersfvNV(enum target, uint buffer, uint index,
                                     sizei count, const float *params);
    void ProgramBufferParametersIivNV(enum target, uint buffer, uint index,
                                      sizei count, const int *params);
    void ProgramBufferParametersIuivNV(enum target, uint buffer, uint index,
                                       sizei count, const uint *params);
    void GetIntegerIndexedvEXT(enum value, uint index, boolean *data);
```

New Tokens

Accepted by the <pname> parameter of GetProgramivARB:

```
      MAX_PROGRAM_PARAMETER_BUFFER_BINDINGS_NV          0x8DA0
      MAX_PROGRAM_PARAMETER_BUFFER_SIZE_NV              0x8DA1
```

Accepted by the <target> parameter of ProgramBufferParametersfvNV, ProgramBufferParametersIivNV, and ProgramBufferParametersIuivNV, BindBufferRangeNV, BindBufferOffsetNV, BindBufferBaseNV, and BindBuffer and the <value> parameter of GetIntegerIndexedvEXT:

```
      VERTEX_PROGRAM_PARAMETER_BUFFER_NV                0x8DA2
      GEOMETRY_PROGRAM_PARAMETER_BUFFER_NV              0x8DA3
      FRAGMENT_PROGRAM_PARAMETER_BUFFER_NV              0x8DA4
```

Additions to Chapter 2 of the OpenGL 2.0 Specification (OpenGL Operation)

Modify "Section 2.14.1" of the ARB_vertex_program specification.

(Add after the discussion of environment parameters.)

Additionally, each program target has an array of parameter buffer binding points, to which a buffer object (Section 2.9) can be bound. The number of available binding points is given by the implementation-dependent constant MAX_PROGRAM_PARAMETER_BUFFER_BINDINGS_NV. These binding points are shared by all programs of a given type. All bindings are initialized to the name zero, which indicates that no valid binding is present.

A program parameter binding is associated with a buffer object using BindBufferOffset with a <target> of VERTEX_PROGRAM_PARAMETER_BUFFER_NV, GEOMETRY_PROGRAM_PARAMETER_BUFFER_NV, or FRAGMENT_PROGRAM_PARAMETER_BUFFER_NV and <index> corresponding to the number of the desired binding point. The error INVALID_VALUE is generated if the value of <index> is greater than or equal to MAX_PROGRAM_PARAMETER_BUFFER_BINDINGS.

Buffer objects are made to be sources of program parameter buffers by calling one of

```
  void BindBufferRangeNV(enum target, uint index, uint buffer,
                         intptr offset, sizeiptr size)
  void BindBufferOffsetNV(enum target, uint index, uint buffer,
                          intptr offset)
  void BindBufferBaseNV(enum target, uint index, uint buffer)
```

where <target> is set to VERTEX_PROGRAM_PARAMETER_BUFFER_NV, GEOMETRY_PROGRAM_PARAMETER_BUFFER_NV, or FRAGMENT_PROGRAM_PARAMETER_BUFFER_NV. Any of the three BindBuffer* commands perform the equivalent of BindBuffer(target, buffer). <buffer> specifies which buffer object to bind to the target at index number <index>. <index> must be less than the value of MAX_PROGRAM_PARAMETER_BUFFER_BINDINGS_NV. <offset> specifies a starting offset into the buffer object <buffer>. <size> specifies the number of elements in the bound portion of the buffer. Both <offset> and <size> are in basic machine units. The error INVALID_VALUE is generated if the value of <size> is less than or equal to zero. The error INVALID_VALUE is generated if <offset> or <size> are not word-aligned. For program parameter buffers, the error INVALID_VALUE is generated if <offset> is non-zero.

BindBufferBaseNV is equivalent to calling BindBufferOffsetNV with an <offset> of 0. BindBufferOffsetNV is the equivalent of calling BindBufferRangeNV with <size> = sizeof(buffer) - <offset> and rounding <size> down so that it is word-aligned.

All program parameter buffer parameters are either single-component 32-bit words or four-component vectors made up of 32-bit words. The program parameter buffers may hold signed integer, unsigned integer, or floating-point data. There is a limit on the maximum number of words of a buffer object that can be accessed using any single parameter buffer binding point, given by the implementation-dependent constant MAX_PROGRAM_PARAMETER_BUFFER_SIZE_NV. Buffer objects larger than this size may be used, but the results of accessing portions of the buffer object beyond the limit are undefined.

The commands

```
  void ProgramBufferParametersfvNV(enum target, uint buffer, uint index,
                                   sizei count, const float *params);
  void ProgramBufferParametersIivNV(enum target, uint buffer, uint index,
                                    sizei count, const int *params);
  void ProgramBufferParametersIuivNV(enum target, uint buffer, uint index,
                                    sizei count, const uint *params);
```

update words <index> through <index>+<count>-1 in the buffer object bound to the binding point numbered <buffer> for the program target <target>. The new data is referenced by <params>. The error INVALID_OPERATION is generated if no buffer object is bound to the binding point numbered <buffer>. The error INVALID_VALUE is generated if <index>+<count> is greater than either the number of words in the buffer object or the maximum parameter buffer size MAX_PROGRAM_PARAMETER_BUFFER_SIZE_NV. These functions perform an operation functionally equivalent to calling BufferSubData, but possibly with higher performance.

Additions to Chapter 3 of the OpenGL 2.0 Specification (Rasterization)

None.

Additions to Chapter 4 of the OpenGL 2.0 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 2.0 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 2.0 Specification (State and State Requests)

Modify the second paragraph of section 6.1.1 (Simple Queries) p. 244 to read as follows:

...<data> is a pointer to a scalar or array of the indicated type in which to place the returned data.

```
    void GetIntegerIndexedvEXT(enum target, uint index,
                               boolean *data);
```

are used to query indexed state. <target> is the name of the indexed state and <index> is the index of the particular element being queried. <data> is a pointer to a scalar or array of the indicated type in which to place the returned data.

Additions to the AGL/GLX/WGL Specifications

None

GLX Protocol

TBD

Dependencies on NV_transform_feedback

Both NV_transform_feedback and this extension define the behavior of BindBuffer{Range, Offset, Base}NV. Both definitions should be functionally identical.

Errors

The error INVALID_VALUE is generated by BindBufferRangeNV, BindBufferOffsetNV, or BindBufferBaseNV if <target> is VERTEX_PROGRAM_PARAMETER_BUFFER_NV, GEOMETRY_PROGRAM_PARAMETER_BUFFER_NV, or FRAGMENT_PROGRAM_PARAMETER_BUFFER_NV, and <index> is greater than or equal to MAX_PROGRAM_PARAMETER_BUFFER_BINDINGS.

The error INVALID_VALUE is generated by BindBufferRangeNV or BindBufferOffsetNV if <offset> or <size> is not word-aligned.

```
    The error INVALID_VALUE is generated by BindBufferRangeNV if <size> is
    less than zero.

    The error INVALID_VALUE is generated by BindBufferRangeNV or
    BindBufferOffsetNV if <target> is VERTEX_PROGRAM_PARAMETER_BUFFER_NV,
    GEOMETRY_PROGRAM_PARAMETER_BUFFER_NV, or
    FRAGMENT_PROGRAM_PARAMETER_BUFFER_NV, and <offset> is non-zero.

    The error INVALID_OPERATION is generated by ProgramBufferParametersfvNV,
    ProgramBufferParametersIivNV, or ProgramBufferParametersIuivNV if no
    buffer object is bound to the binding point numbered <buffer> for program
    target <target>.

    The error INVALID_VALUE is generated by ProgramBufferParametersfvNV,
    ProgramBufferParametersIivNV, or ProgramBufferParametersIuivNV if the sum
    of <index> and <count> is greater than either the number of words in the
    buffer object boudn to <buffer> or the maximum parameter buffer size
    MAX_PROGRAM_PARAMETER_BUFFER_SIZE_NV.
```

New State

    (Modify ARB_vertex_program, Table X.6 -- Program State)

Get Value	Type	Get Command	Initial Value	Description	Sec.	Attribute
VERTEX_PROGRAM_PARAMETER_ BUFFER_NV	Z+	GetIntegerv	0	Active vertex program buffer object binding	2.14.1	-
VERTEX_PROGRAM_PARAMETER_ BUFFER_NV	nxZ+	GetInteger- IndexedvEXT	0	Buffer objects bound for vertex program use	2.14.1	-
GEOMETRY_PROGRAM_PARAMETER_ BUFFER_NV	Z+	GetIntegerv	0	Active geometry program buffer object binding	2.14.1	-
GEOMETRY_PROGRAM_PARAMETER_ BUFFER_NV	nxZ+	GetInteger- IndexedvEXT	0	Buffer objects bound for geometry program use	2.14.1	-
FRAGMENT_PROGRAM_PARAMETER_ BUFFER_NV	Z+	GetIntegerv	0	Active fragment program buffer object binding	2.14.1	-
FRAGMENT_PROGRAM_PARAMETER_ BUFFER_NV	nxZ+	GetInteger- IndexedvEXT	0	Buffer objects bound for fragment program use	2.14.1	-

New Implementation Dependent State

Get Value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
MAX_PROGRAM_PARAMETER_ BUFFER_BINDINGS_NV	Z	GetProgram- ivARB	8	size of program parameter binding tables	2.14.1	-
MAX_PROGRAM_PARAMETER_ BUFFER_SIZE_NV	Z	GetProgram- ivARB	4096	maximum usable size of program parameter buffers	2.14.1	-

Examples

```
    !!NVfp4.0
    # Legal
    BUFFER bones[] = { program.buffer[0] };
    ALIAS funBone = bones[69];
    MOV t, bones[1];
    # Illegal
    ALIAS numLights = program.buffer[5][6];
    MOV t, program.buffer[3][x];
    END
```

Issues

(1) PBO is already taken as an acronym? What do we call this?

RESOLVED: PaBO.

(2) How should the ability to simultaneously access multiple parameter buffers be exposed?

RESOLVED: In the program text (see NV_gpu_program4), the buffers are referred to using a buffer binding statement which is dereferenced in the instructions. In the rest of the APIs, an array of internal binding points is provided, which are dereferenced using the index parameter of BindBufferBase and associated functions.

(3) Should program parameter buffer bindings be provided per-target (i.e., environment parameters), per-program (i.e., local parameters), or some combination of the two?

RESOLVED: Per-target. That fits most naturally with the ARB program model, similar to textures. Having both per-program and per-target add complexity with no benefit.

(4) Should references to the parameter buffer be scalar or vector?

RESOLVED: Scalar. Having vector is more consistent with the legacy APIs, but is more difficult to build the arbitrary data structures that are interesting to store in a parameter buffer. A future extension can define an alternate keyword in the program text to specify accesses of a different size.

(5) Should parameter buffers be editable using the ProgramEnvParameter API?

RESOLVED: No. There is a new parallel API for the bindable buffers, including the ability to update multiple parameters at a time. These are more convenient than having to rebind for BufferData and potentially faster.

(6) Should parameter buffers be editable outside the ProgramBufferParameters API?

RESOLVED: Yes. The use of buffer objects allows the buffers to be naturally manipulated using normal buffer object mechanisms. That includes CPU mapping, loading via BufferData or BufferSubData, and even reading data back using the ARB_pixel_buffer_object extension.

(7) Will buffer object updates from different sources cause potential synchronization problems? If so, how will they be resolved.

RESOLVED: If reads and write occur in the course of the same call (e.g. reading from a buffer using parameter buffer binding while writing to it using transform feedback. All other cases are allowed and occur in command order. Any synchronization is handled by the GL.

(8) Is there an implementation-dependent limit to the size of program parameter buffers?

RESOLVED: Yes, limited-size buffers are provided to reduce the complexity of the GPU design that supports program parameter buffer access and updates. However, the minimum limit is 16K scalar parameters, or 64KB. A larger buffer object can be provided, but only the first 64KB is accessible. The limit is queryable with GetProgramivARB with <pname> MAX_PROGRAM_PARAMETER_BUFFER_SIZE_NV.

(9) With scalar buffers, which parameter setting routines do we need?

UNRESOLVED: A function to set N scalars is very important. It might be nice to have convenience functions that take 1 or 4 parameters directly.

(10) Do we need GetProgramBufferParameter functions?

UNRESOLVED: Probably not - they aren't perf critical and offer no functionality beyond getting the buffer object data any of the standard ways.

(11) What happens if a value written using ProgramBufferParametersfNV is read as an integer or the other way around?

RESOLVED: Undefined - likely just a raw bit cast between whatever internal representations are used by the GL.

Revision History

```
Rev.    Date    Author    Changes
----  --------  --------  ----------------------------------------
 7    04/18/07  pbrown    Fixed state table to include the buffer
                          object binding array for each program type.
```

Name

    NV_pixel_data_range

Name Strings

    GL_NV_pixel_data_range

Notice

    Copyright NVIDIA Corporation, 2000, 2001, 2002.

IP Status

    NVIDIA Proprietary.

Status

    Shipping (version 1.0)

Version

    NVIDIA Date: November 7, 2002 (version 1.0)
    $Id: //sw/main/docs/OpenGL/specs/GL_NV_pixel_data_range.txt#5 $

Number

    284

Dependencies

    Written based on the wording of the OpenGL 1.3 specification.

    If this extension is implemented, the WGL or GLX memory allocator
    interface specified in NV_vertex_array_range must also be
    implemented. Please refer to the NV_vertex_array_range specification
    for further information on this interface.

Overview

    The vertex array range extension is intended to improve the
    efficiency of OpenGL vertex arrays. OpenGL vertex arrays' coherency
    model and ability to access memory from arbitrary locations in memory
    prevented implementations from using DMA (Direct Memory Access)
    operations.

    Many image-intensive applications, such as those that use dynamically
    generated textures, face similar problems. These applications would
    like to be able to sustain throughputs of hundreds of millions of
    pixels per second through DrawPixels and hundreds of millions of
    texels per second through TexSubImage.

    However, the same restrictions that limited vertex throughput also
    limit pixel throughput.

    By the time that any pixel operation that reads data from user memory
    returns, OpenGL requires that it must be safe for the application to

start using that memory for a different purpose. This coherency model prevents asynchronous DMA transfers directly out of the user's buffer.

There are also no restrictions on the pointer provided to pixel operations or on the size of the data. To facilitate DMA implementations, the driver needs to know in advance what region of the address space to lock down.

Vertex arrays faced both of these restrictions already, but pixel operations have one additional complicating factor -- they are bidirectional. Vertex array data is always being transfered from the application to the driver and the HW, whereas pixel operations sometimes transfer data to the application from the driver and HW. Note that the types of memory that are suitable for DMA for reading and writing purposes are often different. For example, on many PC platforms, DMA pulling is best accomplished with write-combined (uncached) AGP memory, while pushing data should use cached memory so that the application can read the data efficiently once it has been read back over the AGP bus.

This extension defines an API where an application can specify two pixel data ranges, which are analogous to vertex array ranges, except that one is for operations where the application is reading data (e.g. glReadPixels) and one is for operations where the application is writing data (e.g. glDrawPixels, glTexSubImage2D, etc.). Each pixel data range has a pointer to its start and a length in bytes.

When the pixel data range is enabled, and if the pointer specified as the argument to a pixel operation is inside the corresponding pixel data range, the implementation may choose to asynchronously pull data from the pixel data range or push data to the pixel data range. Data pulled from outside the pixel data range is undefined, while pushing data to outside the pixel data range produces undefined results.

The application may synchronize with the hardware in one of two ways: by flushing the pixel data range (or causing an implicit flush) or by using the NV_fence extension to insert fences in the command stream.

Issues

* *The vertex array range extension required that all active vertex arrays must be located inside the vertex array range. Should this extension be equally strict?*

  RESOLVED: No, because a user may want to use the pixel data range for one type of operation (say, texture downloads) but still be able to use standard non-PDR pixel operations for everything else. Requiring that apps disable PDR every time such an operation occurs would be burdensome and make it difficult to integrate this extension into a larger app with minimal changes. So, for each pixel operation, we will look at the pointer provided by the application. If it's inside the PDR, the PDR rules apply, and if it's not inside the PDR, it's a standard GL pixel operation, even if some of the data is actually inside the PDR.

* *Reads and writes may require different types of memory. How do we handle this?*

  RESOLVED: The allocator interface already provides the ability to specify different read and write frequencies. A buffer for a write PDR should probably be allocated with a high write frequency and low read frequency, while a read PDR's buffer should have a low write and high read frequency.

  Having two PDRs is essential because a single application may want to perform both asynchronous reads and writes simultaneously.

* *What happens if a PDR pixel operation pulls data from a location outside the PDR?*

  RESOLVED: The data pulled is undefined, and program termination may result.

* *What happens if a PDR pixel operation pushes data to a location outside the PDR?*

  RESOLVED: The contents of that memory location become undefined, and program termination may result.

* *What happens if the hardware can't support the operation?*

  RESOLVED: The operation may be slow, because we may need to, for example, read the pixel data out of uncached memory with the CPU, but it should still work. So this should never be a problem; in fact, it means that a basic implementation that accelerates only, say, one operation is quite trivial.

* *Should there be any limitations to what operations should be supported?*

  RESOLVED: No, in theory any pixel operation that accesses a user's buffer can work with PDR. This includes Bitmap, PolygonStipple, GetTexImage, ConvolutionFilter2D, etc. Many are unlikely to be accelerated, but there is no reason to place arbitrary restrictions. A list of possibly supported operations is provided for OpenGL 1.2.1 with ARB_imaging support and for all the extensions currently supported by NVIDIA. Developers should carefully read the Implementation Details provided by their vendor before using the extension.

* *Should PixelMap and GetPixelMap be supported?*

  RESOLVED: Yes. They're not really pixel path operations, but, again, there is no good reason to omit operations, and they _are_ operations that pass around big chunks of pixel-related data. If we support PolygonStipple, surely we should support this.

* *Can the PDRs and the VAR overlap and/or be the same buffer?*

  RESOLVED: Yes. In fact, it is expected that one of the preferred modes of usage for this extension will be to use the same AGP buffer for both the write PDR and the VAR, so it can be used for both dynamic texturing and dynamic geometry.

* *Can video memory buffers be used?*

  RESOLVED: Yes, assuming the implementation supports using them for PDR. On systems with AGP Fast Writes, this may be interesting in some cases. Another possible use for this is to treat a video memory buffer as an offscreen surface, where DrawPixels can be thought of as a blit from offscreen memory to a GL surface, and ReadPixels can be thought of as a blit from a GL surface to offscreen memory. This technique should be used with caution, because there are other alternatives, such as pbuffers, aux buffers, and even textures.

* *Do we want to support more than one read and one write PDR?*

  RESOLVED: No, but I could imagine uses for it. For example, an app could use two system memory buffers (one read, one write PDR) and a single video memory buffer (both read and write). Do we need a scheme where an unlimited number of PDR buffers can be specified? Ugh. I hope not. I can't think of a good reason to use more than 3 buffers, and even that is stretching it.

* *Do we want a separate enable for both the read and write PDR?*

  RESOLVED: Yes. In theory, they are completely independent, and we should treat them as such.

* *Is there an equivalent to the VAR validity check?*

  RESOLVED: No. When a vertex array call occurs, all the vertex array state is already set. We can know in advance whether all the pointers, strides, etc. are set up in a satisfactory way. However, for a pixel operation, much of the state is provided on the same function call that performs the operation. For example, the pixel format of the data may need to match that of the framebuffer. We can't know this without looking at the format and type arguments.

  An alternative might be some sort of "proxy" mechanism for pixel operations, but this seems to be very complicated.

* *Do we want a more generalized API? What stops us from needing a DMA extension for every single conceivable use in the future?*

  RESOLVED: No, this is good enough. Since new extensions will probably require new semantics anyhow, we'll just live with that. Maybe if the ARB wants to create a more generic "DMA" extension, these issues can be revisited.

```
    *   How do applications synchronize with the hardware?

        RESOLVED: A new command, FlushPixelDataRangeNV, is provided, that
        is analogous to FlushVertexArrayRangeNV.  Applications can also
        use the Finish command.  The NV_fence extension is best for
        applications that need fine-grained synchronization.

    *   Should enabling or disabling a PDR induce an implicit PDR flush?

        RESOLVED: No.  In the VAR extension, enabling and disabling the
        VAR does induce a VAR flush, but this has proven to be more
        problematic than helpful, because it makes it much more difficult
        to switch between VAR and non-VAR rendering; the VAR2 extension
        lifts this restriction, and there is no reason to get this wrong
        a second time.

        The PDR extension does not suffer from the problem of enabling
        and disabling frequently, because non-PDR operations are
        permitted simply by providing a pointer outside of the PDR, but
        there is no clear reason why the enable or disable should cause
        a quite unnecessary PDR flush.

    *   Should this state push/pop?

        RESOLVED: Yes, but via a Push/PopClientAttrib and the
        GL_CLIENT_PIXEL_STORE_BIT bit.  Although this is heavyweight
        state, VAR also allowed push/pop.  It does fit nicely into an
        existing category, too.

    *   Should making another context current cause a PDR flush?

        RESOLVED: No.  There's no fundamental reason it should.  Note
        that apps should be careful to not free their memory until the
        hardware is not using it... note also that this decision is
        inconsistent with VAR, which did guarantee a flush here.

    *   Is the read PDR guaranteed to give you either old or new values,
        or is it truly undefined?

        RESOLVED: Undefined.  This may ease implementation constraints
        slightly.  Apps must not rely at all on the contents of the
        region where the readback is occuring until it is known to be
        finished.

        An example of how an implementation might conceivably require
        this is as follows.  Suppose that a piece of hardware, for some
        reason, can only write full 32-byte chunks of data.  Any bytes
        that were supposed to be unwritten are in fact trashed by the
        hardware, filled with garbage.  By careful fixups (read the
        contents before the operation, restore when done), the driver may
        be able to hide this fact, but a requirement that either new or
        old data must show up would be violated.

        Or, more trivially, you might implement certain pixel operations
        as an in-place postprocess on the returned data.
```

        It is not anticipated that NVIDIA implementations will need this
        flexibility, but it is nevertheless provided.

    *   *How should an application allocate its PDR memory?*

        The app should use wglAllocateMemoryNV, even for a read PDR in
        system memory. Using malloc may result in suboptimal
        performance, because the driver will not be able to choose an
        optimal memory type. For ReadPixels to system memory, you might
        set a read frequency of 1.0, a write frequency of 0.0, and a
        priority of 1.0. The driver might allocate PCI memory, or
        physically contiguous PCI memory, or cachable AGP memory, all
        depending on the performance characteristics of the device.
        While memory from malloc will work, it does not allow the driver
        to make these decisions, and it will certainly never give you AGP
        memory.

        Write PDR memory for purposes of streaming textures, etc. works
        exactly the same as VAR memory for streaming vertices. You can,
        and in fact are encouraged to, use the same circular buffer for
        both vertices and textures.

        If you have different needs (not just streaming textures or
        asynchronous readbacks), you may want your pixel data in video
        memory.

New Procedures and Functions

```
    void PixelDataRangeNV(enum target, sizei length, void *pointer)
    void FlushPixelDataRangeNV(enum target)
```

New Tokens

    Accepted by the <target> parameter of PixelDataRangeNV and
    FlushPixelDataRangeNV, and by the <cap> parameter of
    EnableClientState, DisableClientState, and IsEnabled:

```
        WRITE_PIXEL_DATA_RANGE_NV                       0x8878
        READ_PIXEL_DATA_RANGE_NV                        0x8879
```

    Accepted by the <pname> parameter of GetBooleanv, GetIntegerv,
    GetFloatv, and GetDoublev:

```
        WRITE_PIXEL_DATA_RANGE_LENGTH_NV                0x887A
        READ_PIXEL_DATA_RANGE_LENGTH_NV                 0x887B
```

    Accepted by the <pname> parameter of GetPointerv:

```
        WRITE_PIXEL_DATA_RANGE_POINTER_NV               0x887C
        READ_PIXEL_DATA_RANGE_POINTER_NV                0x887D
```

Additions to Chapter 2 of the OpenGL 1.3 Specification (OpenGL Operation)

    None.

Additions to Chapter 3 of the OpenGL 1.3 Specification (Rasterization)

Add new section to Section 3.6, "Pixel Rectangles", on page 113:

"3.6.7 Write Pixel Data Range Operation

Applications can enhance the performance of DrawPixels and other commands that transfer large amounts of pixel data by using a pixel data range. The command

```
   void PixelDataRangeNV(enum target, sizei length, void *pointer)
```

specifies one of the current pixel data ranges. When the write pixel data range is enabled and valid, pixel data transfers from within the pixel data range are potentially faster. The pixel data range is a contiguous region of (virtual) address space for placing pixel data. The "pointer" parameter is a pointer to the base of the pixel data range. The "length" pointer is the length of the pixel data range in basic machine units (typically unsigned bytes). For the write pixel data range, "target" must be WRITE_PIXEL_DATA_RANGE_NV.

The pixel data range address space region extends from "pointer" to "pointer + length - 1" inclusive.

There is some system burden associated with establishing a pixel data range (typically, the memory range must be locked down). If either the pixel data range pointer or size is set to zero, the previously established pixel data range is released (typically, unlocking the memory).

The pixel data range may not be established for operating system dependent reasons, and therefore, not valid. Reasons that a pixel data range cannot be established include spanning different memory types, the memory could not be locked down, alignment restrictions are not met, etc.

The write pixel data range is enabled or disabled by calling EnableClientState or DisableClientState with the symbolic constant WRITE_PIXEL_DATA_RANGE_NV.

The write pixel data range is valid when the following conditions are met:

- o WRITE_PIXEL_DATA_RANGE_NV is enabled.
- o PixelDataRangeNV has been called with a non-null pointer and non-zero size, for target WRITE_PIXEL_DATA_RANGE_NV.
- o The write pixel data range has been established.
- o An implementation-dependent validity check based on the pointer alignment, size, and underlying memory type of the write pixel data range region of memory.

Otherwise, the write pixel data range is not valid.

The commands, such as DrawPixels, that may be made faster by the write pixel data range are listed in the Appendix.

When the write pixel data range is valid, an attempt will be made to accelerate these commands if and only if the data pointer argument to the command lies within the write pixel data range. No attempt will be made to accelerate commands whose base pointer is outside this range. Accessing data outside the write pixel data range when the base pointer lies within the range and the range is valid will produce undefined results and may cause program termination.

The standard OpenGL pixel data coherency model requires that pixel data be extracted from the user's buffer immediately, before the pixel command returns. When the write pixel data range is valid, this model is relaxed so that changes made to pixel data until the next "write pixel data range flush" may affect pixel commands in non-sequential ways. That is, a call to a pixel command that precedes a change to pixel data (without an intervening "write pixel data range flush") may access the changed data; though a call to a pixel command following a change to pixel data must always access the changed data, and never the original data.

A 'write pixel data range flush' occurs when one of the following operations occur:

- o Finish returns.
- o FlushPixelDataRangeNV (with target WRITE_PIXEL_DATA_RANGE_NV) returns.
- o PixelDataRangeNV (with target WRITE_PIXEL_DATA_RANGE_NV) returns.

The client state required to implement the write pixel data range consists of an enable bit, a memory pointer, and an integer size.

If the memory mapping of pages within the pixel data range changes, using the pixel data range has undefined effects. To ensure that the pixel data range reflects the address space's current state, the application is responsible for calling PixelDataRange again after any memory mapping changes within the pixel data range."

Additions to Chapter 4 of the OpenGL 1.3 Specification (Per-Fragment Operations and the Frame Buffer)

Add new section to Section 4.3, "Pixel Draw/Read State", on page 180:

"4.3.5 Read Pixel Data Range Operation

The read pixel data range is similar to the write pixel data range (see section 3.6.7), but is specified with PixelDataRangeNV with a target READ_PIXEL_DATA_RANGE_NV. It is exactly analogous to the write pixel data range, but applies to commands where OpenGL returns pixel data to the caller, such as ReadPixels. The list of commands to which the read pixel data range applies can be found in the Appendix.

Validity checks and flushes of the read pixel data range behave in a manner exactly analogous to those of the write pixel data range, though any implementation-dependent checks may differ between the two types of pixel data range.

The standard OpenGL pixel data coherency model requires that pixel data be written into the user's buffer immediately, before the pixel command returns. When the read pixel data range is valid, this model is relaxed so that this data may not necessarily be available until the next "read pixel data range flush". Until such point in time, an attempt to read the buffer returns undefined values.

If both the read and write pixel data ranges are valid and overlap, then all operations involving both in the same thread are automatically synchronized. That is, the write pixel data range operation will automatically wait for any pending read pixel data range results to become available before attempting to retrieve them. However, if the operations are performed from different threads, the user is responsible for all such synchronization.

Read pixel data range operations are also synchronized with vertex array range operations in the same way.

The client state required to implement the read pixel data range consists of an enable bit, a memory pointer, and an integer size."

Additions to Chapter 5 of the OpenGL 1.3 Specification (Special Functions)

Add the following to the end of Section 5.4 "Display Lists" (page 179):

"PixelDataRangeNV and FlushPixelDataRangeNV are not complied into display lists but are executed immediately.

If a display list is compiled while WRITE_PIXEL_DATA_RANGE_NV is enabled, all commands affected by that enable are accumulated into a display list as if WRITE_PIXEL_DATA_RANGE_NV is disabled.

The state of the read pixel data range does not affect display list compilation, because those commands that might be accelerated by a read pixel data range are commands that are executed immediately rather than being compiled into a display list (ReadPixels and GetTexImage, for example)."

Additions to Chapter 6 of the OpenGL 1.3 Specification (State and State Requests)

None.

Additions to the GLX Specification

"OpenGL implementations using GLX indirect rendering should fail to set up the pixel data range and will not accelerate any pixel operations using it. Additionally, glXAllocateMemoryNV always fails to allocate memory (returns NULL) when used with an indirect rendering context."

GLX Protocol

None

Errors

INVALID_OPERATION is generated if PixelDataRangeNV or FlushPixelDataRangeNV is called between the execution of Begin and the corresponding execution of End.

INVALID_ENUM is generated if PixelDataRangeNV or FlushPixelDataRangeNV is called when target is not WRITE_PIXEL_DATA_RANGE_NV or READ_PIXEL_DATA_RANGE_NV.

INVALID_VALUE is generated if PixelDataRangeNV is called when length is negative.

New State

Get Value	Get Command	Type	Initial Value	Attrib
WRITE_PIXEL_DATA_RANGE_NV	IsEnabled	B	False	pixel-store
READ_PIXEL_DATA_RANGE_NV	IsEnabled	B	False	pixel-store
WRITE_PIXEL_DATA_RANGE_POINTER_NV	GetPointerv	Z+	0	pixel-store
READ_PIXEL_DATA_RANGE_POINTER_NV	GetPointerv	Z+	0	pixel-store
WRITE_PIXEL_DATA_RANGE_LENGTH_NV	GetIntegerv	Z+	0	pixel-store
READ_PIXEL_DATA_RANGE_LENGTH_NV	GetIntegerv	Z+	0	pixel-store

Appendix: Operations Supported

```
    In unextended OpenGL 1.3 with ARB_imaging support, the following
    commands may take advantage of the write PDR:

        glBitmap
        glColorSubTable
        glColorTable
        glCompressedTexImage1D
        glCompressedTexImage2D
        glCompressedTexImage3D
        glCompressedTexSubImage1D
        glCompressedTexSubImage2D
        glCompressedTexSubImage3D
        glConvolutionFilter1D
        glConvolutionFilter2D
        glDrawPixels
        glPixelMapfv
        glPixelMapuiv
        glPixelMapusv
        glPolygonStipple
        glSeparableFilter2D
        glTexImage1D
        glTexImage2D
        glTexImage3D
        glTexSubImage1D
        glTexSubImage2D
        glTexSubImage3D

    In unextended OpenGL 1.3 with ARB_imaging support, the following
    commands may take advantage of the read PDR:

        glGetColorTable
        glGetCompressedTexImage
        glGetConvolutionFilter
        glGetHistogram
        glGetMinmax
        glGetPixelMapfv
        glGetPixelMapuiv
        glGetPixelMapusv
        glGetPolygonStipple
        glGetSeparableFilter
        glGetTexImage
        glReadPixels
```

    No other extensions shipping in the NVIDIA OpenGL drivers add any
    other new commands that may take advantage of this extension,
    although in a few cases there are new commands that alias to other
    commands that may be accelerated by this extension. These commands
    are:

```
        glCompressedTexImage1DARB (ARB_texture_compression)
        glCompressedTexImage2DARB (ARB_texture_compression)
        glCompressedTexImage3DARB (ARB_texture_compression)
        glCompressedTexSubImage1DARB (ARB_texture_compression)
        glCompressedTexSubImage2DARB (ARB_texture_compression)
        glCompressedTexSubImage3DARB (ARB_texture_compression)
        glColorSubTableEXT (EXT_paletted_texture)
        glColorTableEXT (EXT_paletted_texture)
        glGetCompressedTexImageARB (ARB_texture_compression)
        glTexImage3DEXT (EXT_texture3D)
        glTexSubImage3DEXT (EXT_texture3D)
```

NVIDIA Implementation Details

    In the Release 40 OpenGL drivers, the NV_pixel_data_range extension
    is supported on all GeForce/Quadro-class hardware. The following
    commands may potentially be accelerated in this release:

```
        glReadPixels
        glTexImage2D
        glTexSubImage2D
        glCompressedTexImage2D
        glCompressedTexImage3D
        glCompressedTexSubImage2D
```

    The following type/format/buffer format sets are accelerated for
    glReadPixels:

type	format	buffer format
GL_UNSIGNED_SHORT_5_6_5	GL_RGB	16-bit color (PCs only -- Macs use 555)
GL_UNSIGNED_INT_8_8_8_8_REV	GL_BGRA	32-bit color w/ alpha
GL_UNSIGNED_BYTE	GL_BGRA	32-bit color w/ alpha (little endian only)
GL_UNSIGNED_SHORT	GL_DEPTH_COMPONENT	16-bit depth
GL_UNSIGNED_INT_24_8_NV	GL_DEPTH_STENCIL_NV	24-bit depth, 8-bit stencil

The following internalformat/type/format sets are accelerated for glTex[Sub]Image2D:

internalformat	type	format
GL_LUMINANCE8	GL_UNSIGNED_BYTE	GL_LUMINANCE
GL_INTENSITY8	GL_UNSIGNED_BYTE	GL_LUMINANCE
GL_ALPHA8	GL_UNSIGNED_BYTE	GL_ALPHA
GL_COLOR_INDEX8_EXT	GL_UNSIGNED_BYTE	GL_COLOR_INDEX
GL_RGB5	GL_UNSIGNED_SHORT_5_6_5	GL_RGB
GL_RGB8	GL_UNSIGNED_INT_8_8_8_8_REV	GL_BGRA
GL_RGBA4	GL_UNSIGNED_SHORT_4_4_4_4_REV	GL_BGRA
GL_RGB5_A1	GL_UNSIGNED_SHORT_1_5_5_5_REV	GL_BGRA
GL_RGBA8	GL_UNSIGNED_INT_8_8_8_8_REV	GL_BGRA
GL_DEPTH_COMPONENT16_SGIX	GL_UNSIGNED_SHORT	GL_DEPTH_COMPONENT
GL_DEPTH_COMPONENT24_SGIX	GL_UNSIGNED_INT_24_8_NV	GL_DEPTH_STENCIL_NV

The following internalformat/type/format sets will be accelerated for glTex[Sub]Image2D on little-endian machines only:

internalformat	type	format
GL_LUMINANCE8_ALPHA8	GL_UNSIGNED_BYTE	GL_LUMINANCE_ALPHA
GL_RGB8	GL_UNSIGNED_BYTE	GL_BGRA
GL_RGBA8	GL_UNSIGNED_BYTE	GL_BGRA

All compressed texture formats are supported for glCompressedTex[Sub]Image[2,3]D.

The following restrictions apply to all commands:
- No pixel transfer operations of any kind may be in use.
- The base address of the PDR must be aligned to a 32-byte boundary.
- The data pointer must be aligned to boundaries of the size of one group of pixels. For example, GL_UNSIGNED_SHORT_5_6_5 data must be aligned to 2-byte boundaries, GL_UNSIGNED_INT_24_8_NV data must be aligned to 4-byte boundaries, and GL_BGRA/GL_UNSIGNED_BYTE data must be aligned to 4-byte boundaries (not 1-byte boundaries). Compressed texture data must be aligned to a block boundary.

No additional restrictions apply to glReadPixels or glCompressedTex[Sub]Image[2,3]D.

The following additional restrictions apply to glTex[Sub]Image2D:
- The texture must fit in video memory.
- The texture must have a border size of zero.
- The stride (in bytes) between two lines of source data must not exceed 65535.
- For non-rectangle textures, the width and height of the destination mipmap level must not exceed 2048, nor be below 2; also, the destination mipmap level must not be 2x2 (for 16-bit textures) or 2x2, 4x2, or 2x4 (for 8-bit textures).

Future software releases may increase the number of accelerated commands and the number of accelerated data formats for each command.

Note also that although all of the formats and commands listed are guaranteed to be accelerated, there may be limitations in the actual implementation not as strict as those stated here; for example, some data formats not listed here may turn out to be accelerated. However, it is highly recommended that you stick to the formats and commands listed in this section. In cases where actual restrictions are less strict, future implementations may very well enforce the listed restriction.

It is also possible that some of these restrictions may become _more_ strict on future chips; though at present no such additional restrictions are known to be likely. Such restrictions would likely take the form of more stringent pitch or alignment restrictions, if they proved to be necessary.

In practice, you should expect that several of these restrictions will be more lenient in a future release.

Revision History

November 7, 2002 - Updated implementation details section with most up-to-date rules on PDR usage. Lifted rule that texture downloads must be 2046 pixels in size or smaller. Removed support for 8-bit texture downloads. Increased max TexSubImage pitch to 65535 from 8191.

Name

    NV_point_sprite

Name Strings

    GL_NV_point_sprite

Notice

IP Status

    No known IP issues.

Status

    Shipping (version 1.1)

Version

    NVIDIA Date: March 6, 2003 (version 1.3)
    $Id: //sw/main/docs/OpenGL/specs/GL_NV_point_sprite.txt#14 $

Number

    262

Dependencies

Written based on the wording of the OpenGL 1.3 specification.

Assumes support for the EXT_point_parameters extension.

Overview

Applications such as particle systems usually must use OpenGL quads rather than points to render their geometry, since they would like to use a custom-drawn texture for each particle, rather than the traditional OpenGL round antialiased points, and each fragment in a point has the same texture coordinates as every other fragment.

Unfortunately, specifying the geometry for these quads can be quite expensive, since it quadruples the amount of geometry required, and it may also require the application to do extra processing to compute the location of each vertex.

The goal of this extension is to allow such apps to use points rather than quads. When GL_POINT_SPRITE_NV is enabled, the state of point antialiasing is ignored. For each texture unit, the app can then specify whether to replace the existing texture coordinates with point sprite texture coordinates, which are interpolated across the point. Finally, the app can set a global parameter for the way to generate the R coordinate for point sprites; the R coordinate can either be zero, the input S coordinate, or the input R coordinate. This allows applications to use a 3D texture to represent a point sprite that goes through an animation, with filtering between frames, for example.

Issues

* Should this spec say that point sprites get converted into quads?

  RESOLVED: No, this would make the spec much uglier, because then we'd have to say that polygon smooth and stipple get turned off, etc. Better to provide a formula for computing the texture coordinates and leave them as points.

* How are point sprite texture coordinates computed?

  RESOLVED: They move smoothly as the point moves around on the screen, even though the pixels touched by the point do not. The exact formula is given in the spec. Note that point sprites' T texture coordinate decreases, not increases, with Y; that is, point sprite textures go top-down, not bottom-up.

* How do point sizes for point sprites work?

  RESOLVED: The original NV_point_sprite spec treated point sprites as being sized like aliased points, i.e., integral sizes only. This was a mistake, because it can lead to visible popping artifacts. In addition, it limits the size of points unnecessarily.

  This revised specification treats point sprite sizes more like antialiased point sizes, but with more leniency. Implementations may choose to not clamp the point size to the antialiased point size range. The set of point sprite sizes available must be a superset of the antialiased point sizes. However, whereas antialiased point sizes are all evenly spaced by the point size granularity, point sprites can have an arbitrary set of sizes. This lets implementations use, e.g., floating-point sizes.

  It is anticipated that this behavior change will not cause any problems for compatibility. In fact, it should be beneficial to quality.

* Should there be a way to query the list of supported point sprite sizes?

  RESOLVED: No. If an implementation were to use, say, a single-precision IEEE float to represent point sizes, the list would be rather long.

* Do mipmaps apply to point sprites?

  RESOLVED: Yes. They are similar to quads in this respect.

```
    *   What of this extension's state is per-texture unit and what
        of this extension's state is state is global?

        RESOLVED: The GL_POINT_SPRITE_NV enable and POINT_SPRITE_R_MODE_NV
        state are global.  The COORD_REPLACE_NV state is per-texture unit
        (state set by TexEnv is per-texture unit).

    *   Should we create an entry point for the R mode?

        RESOLVED: No, we take advantage of the existing glPointParameter
        interface.  Unfortunately, EXT_point_parameters does not define a
        PointParameteri entry point.  This extension adds one.  It could
        live without, but it's a little annoying to have to use a float
        interface to specify an enumerant.

        This is definitely not TexEnv state, because it is global, not
        per texture unit.

    *   What should the suffix for PointParameteri[v] be?

        RESOLVED: NV.  This is an NV extension, and therefore any new
        entry points must be NV also.  This is a bit less aesthetically
        pleasing than matching the EXT suffixes of EXT_point_parameters,
        but it is the right thing to do.

    *   Should there be a global on/off switch for point sprites, or
        should the per-unit enable imply that switch?

        RESOLVED: There is a global switch to turn it on and off.  This
        is probably more convenient for both driver and app, and it
        simplifies the spec.

    *   What should the TexEnv mode for point sprites be called?

        RESOLVED: After much deliberation, COORD_REPLACE_NV seems to be
        appropriate.

    *   What is the motivation for each of the three point sprite R
        modes?

        The R mode is most convenient for applications that may already
        be drawing their own "point sprites" by rendering quads.  These
        applications already need to put the R coordinate in R, and they
        do not need to change their code.

        The S mode is most convenient for applications that do not use
        vertex programs, because it allows them to use TexCoord1 rather
        than TexCoord3 to specify their third texture coordinate.  This
        reduces the size of the vertex data.  Applications that use
        vertex programs are largely unaffected by this, because they can
        map the input S texture coordinate into the output R coordinate
        if they so desire.

        The zero mode may allow some applications to more easily obtain
        the behavior they want out of the dot product functionality of
        the NV_texture_shader extension.  It reduces these dot products
        from three-component dot products into two-component dot
```

products. In some implementations, it may also have higher performance than the other modes.

There is no mode corresponding to the T or Q coordinates because we cannot envision a scenario where such modes would be useful.

* What is the interaction with multisample points, which are round?

  RESOLVED: Point sprites are rasterized as squares, even in multisample mode. Leaving them as round points would make the feature useless.

* How does the point sprite extension interact with fragment program extensions (ARB_fragment_program, NV_fragment_program, etc)?

  RESOLVED: The primary issue is how the interpolanted texture coordinate set appears when fragment attribute variables (ARB terminology) or fragment program attribute registers (NV terminology) are accessed.

  When point sprite is enabled and the GL_COORD_REPLACE_NV state for a given texture unit is GL_TRUE, the texture coordinate set for that texture unit is (s,t,r,1) where the point sprite-overriden s, t, and r are described in the amended Section 3.3 below. The important point is that q is forced to 1.

  For fragment program extensions, q cooresponds to the w component of the respective fragment attribute.

* What push/pop attribute bits control the state of this extension?

  RESOLVED: POINT_BIT for all the state. Also ENABLE_BIT for the POINT_SPRITE_NV enable.

New Procedures and Functions

```
void PointParameteriNV(enum pname, int param)
void PointParameterivNV(enum pname, const int *params)
```

New Tokens

Accepted by the <cap> parameter of Enable, Disable, and IsEnabled, by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev, and by the <target> parameter of TexEnvi, TexEnviv, TexEnvf, TexEnvfv, GetTexEnviv, and GetTexEnvfv:

```
    POINT_SPRITE_NV                               0x8861
```

When the <target> parameter of TexEnvf, TexEnvfv, TexEnvi, TexEnviv, GetTexEnvfv, or GetTexEnviv is POINT_SPRITE_NV, then the value of <pname> may be:

```
    COORD_REPLACE_NV                              0x8862
```

When the <target> and <pname> parameters of TexEnvf, TexEnvfv, TexEnvi, or TexEnviv are POINT_SPRITE_NV and COORD_REPLACE_NV respectively, then the value of <param> or the value pointed to by <params> may be:

```
    FALSE
    TRUE
```

Accepted by the <pname> parameter of PointParameteriNV, PointParameterfEXT, PointParameterivNV, PointParameterfvEXT, GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

```
    POINT_SPRITE_R_MODE_NV                        0x8863
```

When the <pname> parameter of PointParameteriNV, PointParameterfEXT, PointParameterivNV, or PointParameterfvEXT is POINT_SPRITE_R_MODE_NV, then the value of <param> or the value pointed to by <params> may be:

```
    ZERO
    S
    R
```

Additions to Chapter 2 of the OpenGL 1.3 Specification (OpenGL Operation)

None.

Additions to Chapter 3 of the OpenGL 1.3 Specification (Rasterization)

Insert the following paragraphs after the second paragraph of section 3.3 (page 63):

"Point sprites are enabled or disabled by calling Enable or Disable with the symbolic constant POINT_SPRITE_NV. The default state is for point sprites to be disabled. When point sprites are enabled, the state of the point antialiasing enable is ignored.

The point sprite R coordinate mode is set with one of the commands

```
  void PointParameter{if}NV(enum pname, T param)
  void PointParameter{if}vNV(enum pname, const T *params)
```

where pname is POINT_SPRITE_R_MODE_NV. The possible values for param are ZERO, S, and R. The default value is ZERO.

The point sprite texture coordinate replacement mode is set with one of the commands

```
  void TexEnv{if}(enum target, enum pname, T param)
  void TexEnv{if}v(enum target, enum pname, const T *params)
```

where target is POINT_SPRITE_NV and pname is COORD_REPLACE_NV. The possible values for param are FALSE and TRUE. The default value for each texture unit is for point sprite texture coordinate replacement to be disabled."

Replace the first two sentences of the fourth paragraph of section 3.3 (page 63) with the following:

"The effect of a point width other than 1.0 depends on the state of point antialiasing and point sprites. If antialiasing and point sprites are disabled, ..."

Replace the first sentences of the sixth paragraph of section 3.3 (page 64) with the following:

"If antialiasing is enabled and point sprites are disabled, ..."

Insert the following paragraphs at the end of section 3.3 (page 66):

"When point sprites are enabled, then point rasterization produces a fragment for each framebuffer pixel whose center lies inside a square centered at the point's (x_w, y_w), with side length equal to the current point size.

All fragments produced in rasterizing a point sprite are assigned the same associated data, which are those of the vertex corresponding to the point, with texture coordinates s, t, and r replaced with s/q, t/q, and r/q, respectively. If q is less than or equal to zero, the results are undefined. However, for each texture unit where COORD_REPLACE_NV is TRUE, these texture coordinates are replaced with point sprite texture coordinates. The s coordinate varies from 0 to 1 across the point horizontally, while the t coordinate varies from 0 to 1 vertically. The r coordinate depends on the value of POINT_SPRITE_R_MODE_NV. If this is set to ZERO, then the r coordinate is set to zero. If it is set to S, then the r coordinate is set to the s texture coordinate before coordinate replacement takes place. If it is set to R, then the r coordinate is set to the r texture coordinate before coordinate replacement takes place.

The following formula is used to evaluate the s and t coordinates:

```
  s = 1/2 + (x_f + 1/2 - x_w) / size
  t = 1/2 - (y_f + 1/2 - y_w) / size
```

where size is the point's size, x_f and y_f are the (integral) window coordinates of the fragment, and x_w and y_w are the exact, unrounded window coordinates of the vertex for the point.

The widths supported for point sprites must be a superset of those supported for antialiased points. There is no requirement that these widths must be equally spaced. If an unsupported width is requested, the nearest supported width is used instead."

Replace the text of section 3.3.1 (page 66) with the following:

"The state required to control point rasterization consists of the floating-point point width, a bit indicating whether or not antialiasing is enabled, a bit indicating whether or not point sprites are enabled, the current value of the point sprite R coordinate mode, and a bit for the point sprite texture coordinate replacement mode for each texture unit."

Replace the text of section 3.3.2 (page 66) with the following:

"If MULTISAMPLE is enabled, and the value of SAMPLE_BUFFERS is one, then points are rasterized using the following algorithm, regardless of whether point antialiasing (POINT_SMOOTH) is enabled or disabled. Point rasterization produces a fragment for each framebuffer pixel with one or more sample points that intersect a region centered at the point's (x_w, y_w). This region is a circle having diameter equal to the current point width if POINT_SPRITE_NV is disabled, or a square with side equal to the current point width if POINT_SPRITE_NV is enabled. Coverage bits that correspond to sample points that intersect the region are 1, other coverage bits are 0. All data associated with each sample for the fragment are the data associated with the point being rasterized, with the exception of texture coordinates when POINT_SPRITE_NV is enabled; these texture coordinates are computed as described in section 3.3.

Point size range and number of gradations are equivalent to those supported for antialiased points when POINT_SPRITE_NV is disabled. The set of point sizes supported is equivalent to those for point sprites without multisample when POINT_SPRITE_NV is enabled."

Additions to Chapter 4 of the OpenGL 1.3 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 1.3 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 1.3 Specification (State and State Requests)

None.

GLX Protocol

Two new GL rendering commands are added. The following commands are sent to the server as part of a glXRender request:

```
    PointParameteriNV
        2           8+4*n           rendering command length
        2           4221            rendering command opcode
        4           ENUM            pname
                    0x8126 n==1     POINT_SIZE_MIN_ARB
                    0x8127 n==1     POINT_SIZE_MAX_ARB
                    0x8128 n==1     POINT_FADE_THRESHOLD_SIZE_ARB
                    0x8863 n==1     POINT_SPRITE_R_MODE_NV
        4           INT32           param
    PointParameterivNV
        2           8+4*n           rendering command length
        2           4222            rendering command opcode
        4           ENUM            pname
                    0x8126 n==1     POINT_SIZE_MIN_ARB
                    0x8127 n==1     POINT_SIZE_MAX_ARB
                    0x8128 n==1     POINT_FADE_THRESHOLD_SIZE_ARB
                    0x8129 n==3     DISTANCE_ATTENUATION_ARB
                    0x8863 n==1     POINT_SPRITE_R_MODE_NV
        4*n         LISTofINT32     params
```

Errors

None.

New State

(table 6.12, p. 220)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
POINT_SPRITE_NV	B	IsEnabled	False	point sprite enable	3.3	point/enable
POINT_SPRITE_R_MODE_NV	Z3	GetIntegerv	ZERO	R coordinate mode	3.3	point
COORD_REPLACE_NV	2* x B	GetTexEnviv	False	coordinate replacement enable	3.3	point

(table 6.17, p. 225)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
COORD_REPLACE_NV	2* x B	GetTexEnviv	False	coordinate replacement enable	3.3	point

NVIDIA Implementation Details

This extension was first supported for GeForce4 Ti only in NVIDIA's Release 25 drivers. Future drivers will support this extension on all GeForce products.

However, the extension is only hardware-accelerated on the GeForce3 and GeForce4 Ti platforms. In addition, there are restrictions on the cases that are accelerated on the GeForce3.

In order to ensure that you get hardware acceleration on GeForce3, make sure that:

1. The point sprite R mode is set to GL_ZERO. (This is the default.)
2. Coordinate replacement is turned on for texture unit 3 and for no other texture units. This is non-obvious; using texture unit zero will _not_ be accelerated. Also, if coordinate replacement is off for _all_ texture units, that's also unaccelerated.

So, in the typical usage case where you just want a single texture on some points, you should enable TEXTURE_2D on unit 3 but disable it on unit zero.

The GeForce4 Ti platform supports point sprites as large as 8192, but the spacing between sizes becomes larger as the size increases. All other platforms do not support point sprite sizes above 64.

ATI Implementation Details

This extension is supported on the Radeon 8000 series and later platforms.

In order to ensure that Radeon 8000 series will accelerate point sprite rendering using TCL hardware, make sure that the point sprite R mode is set to GL_ZERO. (This is the default.)

Radeon 8000 series can render points as large as 2047.

Revision History

June 4, 2002 - Added implementation details section. Fixed a typo in the overview. Changed behavior of point sizes so that fractional sizes are allowed and so that implementations can support large point sprites or use floating-point point size representations. Significant rewrite of spec language to cover this new point size behavior.

Name

    NV_present_video

Name Strings

    GL_NV_present_video
    GLX_NV_present_video
    WGL_NV_present_video

Status

    Implemented in 165.33 driver for NVIDIA SDI devices.

Version

    Last Modified Date: February 20, 2008
    Author Revision: 6
    $Date$ $Revision$

Number

    347

Dependencies

    OpenGL 1.1 is required.

    ARB_occlusion_query is required.
    EXT_timer_query is required.
    ARB_texture_compression affects the definition of this extension.
    ARB_texture_float affects the definition of this extension.
    GLX_NV_video_out affects the definition of this extension.
    EXT_framebuffer_object affects the definition of this extension.
    WGL_ARB_extensions_string affects the definition of this extension.
    WGL_NV_video_out affects the definition of this extension.

    This extension is written against the OpenGL 2.1 Specification
    and the GLX 1.4 Specification.

Overview

    This extension provides a mechanism for displaying textures and
    renderbuffers on auxiliary video output devices. It allows an
    application to specify separate buffers for the individual
    fields used with interlaced output. It also provides a way
    to present frames or field pairs simultaneously in two separate
    video streams. It also allows an application to request when images
    should be displayed, and to obtain feedback on exactly when images
    are actually first displayed.

    This specification attempts to avoid language that would tie it to
    any particular hardware or vendor. However, it should be noted that
    it has been designed specifically for use with NVIDIA SDI products
    and the features and limitations of the spec compliment those of
    NVIDIA's line of SDI video output devices.

New Procedures and Functions

```
    void PresentFrameKeyedNV(uint video_slot,
                             uint64EXT minPresentTime,
                             uint beginPresentTimeId,
                             uint presentDurationId,
                             enum type,
                             enum target0, uint fill0, uint key0,
                             enum target1, uint fill1, uint key1);

    void PresentFrameDualFillNV(uint video_slot,
                                uint64EXT minPresentTime,
                                uint beginPresentTimeId,
                                uint presentDurationId,
                                enum type,
                                enum target0, uint fill0,
                                enum target1, uint fill1,
                                enum target2, uint fill2,
                                enum target3, uint fill3);

    void GetVideoivNV(uint video_slot, enum pname, int *params);
    void GetVideouivNV(uint video_slot, enum pname, uint *params);
    void GetVideoi64vNV(uint video_slot, enum pname, int64EXT *params);
    void GetVideoui64vNV(uint video_slot, enum pname,
                         uint64EXT *params);
    void VideoParameterivNV(uint video_slot, enum pname,
                            const int *params);


    unsigned int *glXEnumerateVideoDevicesNV(Display *dpy, int screen,
                                             int *nelements);
    int glXBindVideoDeviceNV(Display *dpy, unsigned int video_slot,
                             unsigned int video_device,
                             const int *attrib_list);


    DECLARE_HANDLE(HVIDEOOUTPUTDEVICENV);

    int wglEnumerateVideoDevicesNV(HDC hDc,
                                   HVIDEOOUTPUTDEVICENV *phDeviceList);
    BOOL wglBindVideoDeviceNV(HDC hDc, unsigned int uVideoSlot,
                              HVIDEOOUTPUTDEVICENV hVideoDevice,
                              const int *piAttribList);
    BOOL wglQueryCurrentContextNV(int iAttribute, int *piValue);
```

New Tokens

Accepted by the <type> parameter of PresentFrameKeyedNV and PresentFrameDualFillNV:

```
        FRAME_NV                        0x8E26
        FIELDS_NV                       0x8E27
```

```
    Accepted by the <pname> parameter of GetVideoivNV, GetVideouivNV,
    GetVideoi64vNV, GetVideoui64vNV:

        CURRENT_TIME_NV                 0x8E28
        NUM_FILL_STREAMS_NV             0x8E29

    Accepted by the <target> parameter of GetQueryiv:

        PRESENT_TIME_NV                 0x8E2A
        PRESENT_DURATION_NV             0x8E2B

    Accepted by the <attribute> parameter of glXQueryContext:

        GLX_NUM_VIDEO_SLOTS_NV          0x20F0

    Accepted by the <iAttribute> parameter of wglQueryCurrentContextNV:

        WGL_NUM_VIDEO_SLOTS_NV          0x20F0
```

Additions to Chapter 2 of the OpenGL 2.1 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the OpenGL 2.1 Specification (Rasterization)

None

Additions to Chapter 4 of the OpenGL 2.1 Specification (Per-Fragment Operations and the Framebuffer)

Add a new section after Section 4.4:

"4.5 Displaying Buffers

"To queue the display of a set of textures or renderbuffers on one of the current video output devices, call one of:

```
        void PresentFrameKeyedNV(uint video_slot,
                                 uint64EXT minPresentTime,
                                 uint beginPresentTimeId,
                                 uint presentDurationId,
                                 enum type,
                                 enum target0, uint fill0, uint key0,
                                 enum target1, uint fill1, uint key1);

        void PresentFrameDualFillNV(uint video_slot,
                                    uint64EXT minPresentTime,
                                    uint beginPresentTimeId,
                                    uint presentDurationId,
                                    enum type,
                                    enum target0, uint fill0,
                                    enum target1, uint fill1,
                                    enum target2, uint fill2,
                                    enum target3, uint fill3);
```

"PresentFrameKeyedNV can only be used when one output stream is being used for color data. Key data will be presented on the second output stream. PresentFrameDualFillNV can be used only when two output streams are being used for color data. It will present separate color images on each stream simultaneously.

"The <video_slot> parameter specifies which video output slot in the current context this frame should be presented on. If no video output device is bound at <video_slot> at the time of the call, INVALID_OPERATION is generated.

"The value of <minPresentTime> can be set to either the earliest time in nanoseconds that the frame should become visible, or the special value 0. Frame presentation is always queued until the video output's vertical blanking period. At that time, the video output device will consume the frames in the queue in the order they were queued until it finds a frame qualified for display. A frame is qualified if it meets one of the following criteria:

1) The frame's minimum presentation time is the special value zero.

2) The frame's minimum presentation time is less than or equal to the current time and the next queued frame, if it exists, has a minimum presentation time greater than the current time.

Any consumed frames not displayed are discarded. If no qualified frames are found, the current frame continues to display.

"If <beginPresentTimeId> or <presentDurationId> are non-zero, they must name valid query objects (see section 4.1.7, Asynchronous Queries). The actual time at which the video output device began displaying this frame will be stored in the object referred to by <beginPresentTimeId>. The present frame operations will implicitly perform the equivalent of:

```
    BeginQuery(PRESENT_TIME_NV, <beginPresentTimeId>);
    BeginQuery(PRESENT_DURATION_NV, <presentDurationId>);
```

when the respective query object names are valid, followed by the actual present operation, then an implicit EndQuery() for each query started. The result can then be obtained asynchronously via the GetQueryObject calls with a <target> of PRESENT_TIME_NV or PRESENT_DURATION_NV. The results of a query on the PRESENT_TIME_NV target will be the time in nanoseconds when the frame was first started scanning out, and will become available at that time. The results of a query on the PRESENT_DURATION_NV target will be the number of times this frame was fully scanned out by the video output device and will become available when the subsequent frame begins scanning out.

"If the frame was removed from the queue without being displayed, the present duration will be zero, and the present time will refer to the time in nanoseconds when the first subsequent frame that was not skipped began scanning out.

"The query targets PRESENT_TIME_NV and PRESENT_DURATION_NV may not be explicitly used with BeginQuery or EndQuery. Attempting to do so will generate INVALID_ENUM.

```
    "The parameters <type>, <target0>, <fill0>, <key0>, <target1>,
    <fill1>, and <key1> define the data to be displayed on the first
    video output stream.  Valid values for <type> are FIELDS_NV or
    FRAME_NV.  Other values will generate INVALID_ENUM.  The <target0>
    and <target1> parameters can each be one of TEXTURE_2D,
    TEXTURE_RECTANGLE, RENDERBUFFER_EXT, or NONE.  Other values will
    generate INVALID_ENUM.  The <fill0> and <fill1> parameters then name
    an object of the corresponding type from which the color data will
    be read.  Similarly, <key0> and <key1> name an object from which key
    channel data will be read.  If <type> is FIELDS_NV <target0> and
    <target1> can not be NONE and <fill0>, and <fill1> must both name
    valid image objects or INVALID_VALUE is generated.  If <type> is
    FRAME_NV <target0> can not be NONE and <fill0> must name a valid
    object or INVALID_VALUE is generated.  Additionally, <target1> must
    be NONE or INVALID_ENUM is generated.  The values of <fill1> and
    <key1> are ignored.

    "A texture object is considered a valid color image object only if
    it is consistent and has a supported internal format.  A
    renderbuffer object is considered a valid image object if its
    internal format has been specified as one of those supported.
    Implementations must support at least the following internal formats
    for presenting color buffers:

        RGB
        RGBA
        RGB16F_ARB
        RGBA16F_ARB
        RGB32F_ARB
        RGBA32F_ARB
        LUMINANCE
        LUMINANCE_AlPHA

    If no separate key object is specified when using a key output
    stream, the key data is taken from the alpha channel of the color
    object if it is present, or is set to 1.0 otherwise.
    Implementations must support at least the following internal formats
    when presenting key stream buffers:

        RGBA
        RGBA16F_ARB
        RGBA32F_ARB
        LUMINANCE_AlPHA
        DEPTH_COMPONENT

    "The key values are read from the alpha channel unless a depth
    format is used.  For depth formats, the key value is the depth
    value.

    "It is legal to use the same image for more than one of <fill0>,
    <fill1>, <key0>, and <key1>.

    "In the following section, which discusses image dimension
    requirements, the image objects named by <fill0> and <key0> are
    collectively referred to as 'image 0' and the image objects named by
    <fill1> and <key1> are collectively referred to as 'image 1'.  The
    dimensions of a pair of fill and key images must be equal.  If using
    PresentFrameDualFillNV, 'image 0' refers only to <fill0>, and
    'image 1' refers only to <fill1>.

    "If <type> is FRAME_NV image 1 must have a height equal to the
    number of lines displayed per frame on the output device and a width
    equal to the number of pixels per line on the output device or
    INVALID_VALUE will be generated.  Each line in the image will
    correspond to a line displayed on the output device.

    "If <type> is FIELDS_NV, the way in which lines from the image are
    displayed depends on the image's size.  If progressive output is in
    use, image 0 and image 1 must either both have a height equal to the
    number of lines displayed per frame, or both have a height equal to
    the ceiling of half the number of lines displayed per frame.  If an
    interlaced output is in use, the images must either both have a
    height equal to the number of lines displayed per frame, or image 0
    must have a height equal to the number of lines in field one and
    image 1 must have a height equal to the number of lines in field
    two.  The images must both have a width equal to the number of
    pixels per line on the output device.  If any of these conditions
    are not met, INVALID_VALUE is generated.

    "If progressive output is used, the lines are displayed as follows:
    If the images are the same height as a frame, the resulting frame
    displayed is comprised of the first line of image 0, followed by
    the second line of image 1, followed by the third line of image 0,
    and so on until all the lines of a frame have been displayed.  If
    the images are half the height of the frame, the resulting frame
    displayed is comprised of the first line of image 0, followed by the
    first line of image 1, followed by the second line of image 0, and
    so on until the number of lines per frame has been displayed.

    "If interlaced output is used and the images are the same height as
    a frame, the order in which lines are chosen from the images
    depends on the video output mode in use.  If the video output mode
    specifies field 1 as containing the first line of the display, the
    first line of field 1 will come from the first line of image 0,
    followed by the third line from image 0, and so on until the entire
    first field has been displayed.  The first line of field 2 will come
    from the second line of image 1, followed by the fourth line of
    image 1, and so on until the entire second field is displayed.  If
    the mode specifies field 1 as containing the second line of the
    display, the first line of field 1 will come from the second line of
    image 0, followed by the fourth line of image 0, and so on until the
    entire first field is displayed.  The first line of field 2 will
    come from the first line of image 1, followed by the third line of
    image 1, and so on until the entire second field is displayed.

    "If interlaced output is used and the images are the same height as
    individual fields, the order of lines used does not depend on the
    mode in use.  Regardless of the mode used the first line of the
    first field will come from the first line of image 0, followed by
    the second line of image 0, and so on until the entire first field
    has been displayed.  The first line of the second field will come
    from the first line of image 1, followed by the second line of
    image 1, and so on until the entire second field has been displayed.
```

"The parameters <target2>, <fill2>, <target3>, and <fill3> are used identically to <target0>, <fill0>, <target1>, and <fill1> respectively, but they operate on the second color video output stream.

"If the implementation requires a copy as part of the present frame operation, the copy will be transparent to the user and as such will bypass the fragment pipeline completely and will not alter any GL state."

Additions to Chapter 5 of the OpenGL 2.1 Specification (Special Functions)

(Add to section 5.4, "Display Lists", page 244, in the list of commands that are not compiled into display lists)

"Display commands: PresentFrameKeyedNV, PresentFrameDualFillNV

Additions to Chapter 6 of the OpenGL 2.1 Specification (State and State Requests)

(In section 6.1.12, Asynchronous Queries, add the following after paragraph 6, p. 254)

For present time queries (PRESENT_TIME_NV), if the minimum number of bits is non-zero, it must be at least 64.

For present duration queries (PRESENT_DURATION_NV, if the minimum number of bits is non-zero, it must be at least 1.

(Replace section 6.1.15, Saving and Restoring State, p. 264)

Section 6.1.15, Video Output Queries

Information about a video slot can be queried with the commands

```
    void GetVideoivNV(uint video_slot enum pname, int *params);
    void GetVideouivNV(uint video_slot enum pname, uint *params);
    void GetVideoi64vNV(uint video_slot enum pname,
                        int64EXT *params);
    void GetVideoui64vNV(uint video_slot enum pname,
                         uint64EXT *params);
```

If <video_slot> is not a valid video slot in the current context or no video output device is currently bound at <video_slot> an INVALID_OPERATION is generated. If <pname> is CURRENT_TIME_NV, the current time on the video output device in nanoseconds is returned in <params>. If the time value can not be expressed without using more bits than are available in <params>, the value is truncated. If <pname> is NUM_FILL_STREAMS_NV, the number of active video output streams is returned in <params>.

Additions to Appendix A of the OpenGL 2.1 Specification (Invariance)

None

Additions to the WGL Specification

Add a new section "Video Output Devices"

"WGL video output devices can be used to display images with more fine-grained control over the presentation than wglSwapBuffers allows. Use

```
    int wglEnumerateVideoDevicesNV(HDC hDc,
                                   HVIDEOOUTPUTDEVICENV *phDeviceList);
```

to enumerate the available video output devices.

"This call returns the number of video devices available on <hDC>. If <phDeviceList> is non-NULL, an array of valid device handles will be returned in it. The function will assume <phDeviceList> is large enough to hold all available handles so the application should take care to first query the number of devices present and allocate an appropriate amount of memory.

"To bind a video output device to the current context, use

```
    BOOL wglBindVideoDeviceNV(HDC hDc, unsigned int uVideoSlot,
                              HVIDEOOUTPUTDEVICENV hVideoDevice,
                              const int *piAttribList);
```

"wglBindVideoDeviceNV binds the video output device specified by <hVideoDevice> to one of the context's available video output slots specified by <uVideoSlot>. <piAttribList> is a set of attribute name-value pairs that affects the bind operation. Currently there are no valid attributes so <piAttribList> must be either NULL or an empty list. To release a video device without binding another device to the same slot, call wglBindVideoDeviceNV with <hVideoDevice> set to INVALID_HANDLE_VALUE. The bound video output device will be enabled before wglBindVideoDeviceNV returns. It will display black until the first image is presented on it. The previously bound video device, if any, will also be deactivated before wglBindVIdeoDeviceNV resturns. Video slot 0 is reserved for the GL. If wglBindVideoDeviceNV is called with <uVideoSlot> less than 1 or greater than the maximum number of video slots supported by the current context, if <hVideoDevice> does not refer to a valid video output device, or if there is no current context, FALSE will be returned. A return value of TRUE indicates a video device has successfully been bound to the video slot.

Add section "Querying WGL context attributes"

To query an attribute associated with the current WGL context, use

```
    BOOL wglQueryCurrentContextNV(int iAttribute, int *piValue);
```

wglQueryCurrentContextNV will place the value of the attribute named by <iAttribute> in the memory pointed to by <piValue>. If there is no context current or <iAttribute> does not name a valid attribute, FALSE will be returned and the memory pointed to by <piValue> will not be changed. Currently the only valid attribute name is

    WGL_NUM_VIDEO_SLOTS_NV. This attribute contains the number of valid
    video output slots in the current context.

Additions to Chapter 2 of the GLX 1.4 Specification (GLX Operation)

    None

Additions to Chapter 3 of the GLX 1.4 Specification (Functions and Errors)

    Modify table 3.5:

Attribute	Type	Description
GLX_FBCONFIG_ID	XID	XID of GLXFBConfig associated with context
GLX_RENDER_TYPE	int	type of rendering supported
GLX_SCREEN	int	screen number
GLX_NUM_VIDEO_SLOTS_NV	int	number of video output slots this context supports

    Add a section between Sections 3.3.10 and 3.3.11:

    3.3.10a Video Output Devices

    "GLX video output devices can be used to display images with more
    fine-grained control over the presentation than glXSwapBuffers
    allows. Use

```
        unsigned int *glXEnumerateVideoDevicesNV(Display *dpy,
                                                 int screen,
                                                 int *nElements);
```

    to enumerate the available video output devices.

    "This call returns an array of unsigned ints. The number of
    elements in the array is returned in nElements. Each entry in the
    array names a valid video output device. Use XFree to free the
    memory returned by glXEnumerateVideoDevicesNV.

    "To bind a video output device to the current context, use

```
        Bool glXBindVideoDeviceNV(Display *dpy,
                                  unsigned int video_slot,
                                  unsigned int video_device,
                                  const int *attrib_list);
```

    "glXBindVideoDeviceNV binds the video output device specified
    by <video_device> to one of the context's available video
    output slots specified by <video_slot>. <attrib_list> is a
    set of attribute name-value pairs that affects the bind
    operation. Currently there are no valid attributes so <attrib_list>
    must be either NULL or an empty list. To release a video device
    without binding another device to the same slot, call
    glXBindVideoDeviceNV with <video_device> set to "0". Video slot 0
    is reserved for the GL. The bound video output device will be
    enabled before glXBindVideoDeviceNV returns. It will display black
    until the first image is presented on it. The previously bound
    video device, if any, will also be deactivated before

```
    glXBindVIdeoDeviceNV resturns.  If glXBindVideoDeviceNV is called
    with <video_slot> less than 1 or greater than the maximum number of
    video slots supported by the current context, BadValue is generated.
    If <video_device> does not refer to a valid video output device,
    BadValue is generated.  If <attrib_list> contains an invalid
    attribute or an invalid attribute value, BadValue is generated.  If
    glXBindVideoDeviceNV is called without a current context,
    GLXBadContext is generated.
```

Additions to Chapter 4 of the GLX 1.4 Specification (Encoding on the X Byte Stream)

    None

Additions to Chapter 5 of the GLX 1.4 Specification (Extending OpenGL)

    None

Additions to Chapter 6 of the GLX 1.4 Specification (GLX Versions)

    None

GLX Protocol

BindVideoDeviceNV

```
        1       CARD8                   opcode (X assigned)
        1       17                      GLX opcode (glXVendorPrivateWithReply)
        2       6+n                     request length
        4       1332                    vendor specific opcode
        4       CARD32                  context tag
        4       CARD32                  video_slot
        4       CARD32                  video_device
        4       CARD32                  num_attribs
        4*n     LISTofATTRIBUTE_PAIR    attribute, value pairs
      =>
        1       CARD8                   reply
        1                               unused
        2       CARD16                  sequence number
        4       0                       reply length
        4       CARD32                  status
        20                              unused
```

EnumerateVideoDevicesNV

```
        1       CARD8                   opcode (X assigned)
        1       17                      GLX opcode (glXVendorPrivateWithReply)
        2       4                       request length
        4       1333                    vendor specific opcode
        4                               unused
        4       CARD32                  screen
      =>
        1       CARD8                   reply
        1                               unused
        2       CARD16                  sequence number
        4       n                       reply length
        4       CARD32                  num_devices
        4*n     LISTofCARD32            device names
```

```
    PresentFrameKeyedNV
        1       CARD8                   opcode (X assigned)
        1       16                      GLX opcode (glXVendorPrivate)
        2       15                      request length
        4       1334                    vendor specific opcode
        4       CARD32                  context tag
        8       CARD64                  minPresentTime
        4       CARD32                  video_slot
        4       CARD32                  beginPresentTimeId
        4       CARD32                  presentDurationId
        4       CARD32                  type
        4       CARD32                  target0
        4       CARD32                  fill0
        4       CARD32                  key0
        4       CARD32                  target1
        4       CARD32                  fill1
        4       CARD32                  key1

    PresentFrameDualFillNV
        1       CARD8                   opcode (X assigned)
        1       16                      GLX opcode (glXVendorPrivate)
        2       17                      request length
        4       1335                    vendor specific opcode
        4       CARD32                  context tag
        8       CARD64                  minPresentTime
        4       CARD32                  video_slot
        4       CARD32                  beginPresentTimeId
        4       CARD32                  presentDurationId
        4       CARD32                  type
        4       CARD32                  target0
        4       CARD32                  fill0
        4       CARD32                  target1
        4       CARD32                  fill1
        4       CARD32                  target2
        4       CARD32                  fill2
        4       CARD32                  target3
        4       CARD32                  fill3
    GetVideoivNV
        1       CARD8                   opcode (X assigned)
        1       17                      GLX opcode (glXVendorPrivateWithReply)
        2       4                       request length
        4       1336                    vendor specific opcode
        4       CARD32                  context tag
        4       CARD32                  video_slot
        4       CARD32                  pname
      =>
        1       CARD8                   reply
        1                               unused
        2       CARD16                  sequence number
        4       m                       reply length, m = (n==1 ? 0 : n)
        4                               unused
        4       CARD32                  n

        if (n=1) this follows:

        4       INT32                   params
        12                              unused

        otherwise this follows:

        16                              unused
        n*4     LISTofINT32             params

    GetVideouivNV
        1       CARD8                   opcode (X assigned)
        1       17                      GLX opcode (glXVendorPrivateWithReply)
        2       4                       request length
        4       1337                    vendor specific opcode
        4       CARD32                  context tag
        4       CARD32                  video_slot
        4       CARD32                  pname
      =>
        1       CARD8                   reply
        1                               unused
        2       CARD16                  sequence number
        4       m                       reply length, m = (n==1 ? 0 : n)
        4                               unused
        4       CARD32                  n

        if (n=1) this follows:

        4       CARD32                  params
        12                              unused

        otherwise this follows:

        16                              unused
        n*4     LISTofCARD32            params
    GetVideoi64vNV
        1       CARD8                   opcode (X assigned)
        1       17                      GLX opcode (glXVendorPrivateWithReply)
        2       4                       request length
        4       1338                    vendor specific opcode
        4       CARD32                  context tag
        4       CARD32                  video_slot
        4       CARD32                  pname
      =>
        1       CARD8                   reply
        1                               unused
        2       CARD16                  sequence number
        4       m                       reply length, m = (n==1 ? 0 : n)
        4                               unused
        4       CARD32                  n

        if (n=1) this follows:

        8       INT64                   params
        8                               unused

        otherwise this follows:

        16                              unused
        n*8     LISTofINT64EXT          params

    GetVideoui64vNV
        1       CARD8                   opcode (X assigned)
        1       17                      GLX opcode (glXVendorPrivateWithReply)
        2       4                       request length
        4       1339                    vendor specific opcode
        4       CARD32                  context tag
        4       CARD32                  video_slot
        4       CARD32                  pname
      =>
        1       CARD8                   reply
        1                               unused
        2       CARD16                  sequence number
        4       m                       reply length, m = (n==1 ? 0 : n)
        4                               unused
        4       CARD32                  n

        if (n=1) this follows:

        8       CARD64                  params
        8                               unused

        otherwise this follows:

        16                              unused
        n*8     LISTofCARD64            params
```

Dependencies on ARB_occlusion_query:

The generic query objects introduced in ARB_occlusion_query are used as a method to asynchronously deliver timing data to the application. The language describing BeginQueryARB and

EndQueryARB API is also relevant as the same operations are implicitly performed by PresentFrameKeyedNV and PresentFrameDualFillNV.

Dependencies on EXT_timer_query:

The 64-bit types introduced in EXT_timer_query are used in this extension to specify time values with nanosecond accuracy.

Dependencies on ARB_texture_float

If ARB_texture_float is not supported, the floating point internal formats are removed from the list of internal formats required to be supported by the PresentFrame functions.

Dependencies on EXT_framebuffer_object:

If EXT_framebuffer_object is not supported, all references to targets of type RENDERBUFFER_EXT should be removed from the spec language.

Dependencies on GLX_NV_video_out:

Video output resources can not be used simultaneously with this extension and GLX_NV_video_out. If an application on the system has obtained a video device handle from GLX_NV_video_out, no other application may bind any video out devices using this spec until all GLX_NV_video_out devices have been released. Similarly, if an application has bound a video out device using this spec, no other applications on the system can obtain a GLX_NV_video_out device handle until all devices have been unbound.

Dependencies on WGL_ARB_extensions_string:

Because there is no way to extend wgl, these calls are defined in the ICD and can be called by obtaining the address with wglGetProcAddress. The WGL extension string is not included in the GL_EXTENSIONS string. Its existence can be determined with the WGL_ARB_extensions_string extension.

Dependencies on WGL_NV_video_out:

Video output resources can not be used simultaneously with this extension and WGL_NV_video_out. If an application on the system has obtained a video device handle from WGL_NV_video_out, no other application may bind any video out devices using this spec until all WGL_NV_video_out devices have been released. Similarly, if an application has bound a video out device using this spec, no other applications on the system can obtain a WGL_NV_video_out device handle until all devices have been unbound.

Errors

TBD

New State

Get Value	Type	Get Command	Init. Value	Description	Sec	Attribute
CURRENT_QUERY	4xZ+	GetQueryiv	0	Active query object name (occlusion, timer, present time, and present duration)	4.1.7	-
QUERY_RESULT	4xZ+	GetQueryObjectiv	0	Query object result (samples passed, time elapsed, present time, or present duration)	4.1.7	-
QUERY_RESULT_AVAILABLE	4xB	GetQueryObjectiv	TRUE	Query object result available?	4.1.7	-
CURRENT_TIME_NV	1xZ	GetVideoui64vNV	0	Video device timer	4.4	-

New Implementation Dependent state

Get Value	Type	Get Command	Minimum Value	Description	Sec	Attribute
NUM_FILL_STREAMS_NV	1xZ	GetVideouivNV	0	Number of video streams active on a video slot	4.4	-
NUM_VIDEO_SLOTS_NV	1xZ	GetIntegerv	1	Number of video slots a context supports.	4.4	-
QUERY_COUNTER_BITS	4xZ+	GetQueryiv	see 6.1.12	Asynchronous query counter bits (occlusion, timer, present time and present duration queries)	6.1.12	-

Issues

1) How does the user enumerate video devices?

RESOLVED: There will be OS-specific functions that will enumerate OS-specific identifiers that refer to video devices. On WGL, this will likely be tied to an hDC. GPU affinity can then be used to enumerate SDI devices even on GPUs that are not used as part of the windows desktop. On GLX, SDI devices can be enumerated per X screen.

2) How does the user specify data for the second output?

RESOLVED: There will be a separate entry point that accepts up to 4 buffers total.

3) When is SDI output actually enabled?

RESOLVED: The BindVideoDevice functions will enable and disable SDI output.

4) Should the PresentFrame functions return the frame count/identifier?

RESOLVED: No. PresentFrame will instead accept two query object IDs and will implicitly begin and end a query on each of these objects. The first object's query target will be PRESENT_TIME_EXT. Its result will be the time in nanoseconds when the frame was first displayed, and will become available when the frame begins displaying or when a subsequent frame begins displaying if this frame be skipped. The second object's query target will be PRESENT_LENGTH_EXT. The result will be the number of full-frame vblanks that occurred while the frame was displayed. This result will become available when the next frame begins displaying. If the frame was skipped, this value will be 0 and the PRESENT_TIME_EXT result will refer to the time when the first subsequent frame that was not skipped began displaying.

5) Should there be any other queryable video output device attributes?

RESOLVED: There are none. The glXQueryVideoDeviceNV and wglQueryVideoDeviceNV calls have been removed from this specification. They can be added in a separate extension if they are ever needed.

6) Should this spec require a timed present mechanism?

RESOLVED: Yes, this spec will include a mechanism for presenting frames at a specified absolute time and a method for querying when frames were displayed to allow apps to adjust their rendering time. Leaving this out would weaken the PresentFrame mechanism considerably.

7) Should this specification allow downsampling as part of the present operation.

RESOLVED: No, this functionality can retroactively be added to the PresentFrame functions as part of a later spec if necessary.

8) What happens when two outputs are enabled but only one output's worth of buffers are specified?

RESOLVED: This will be an invalid operation. If two outputs are enabled, data must be presented on both of them for every frame.

9) What section of the spec should the PresentFrame functions be in?

RESOLVED: A new section has been added to Chapter 4 to describe functions that control the displaying of buffers.

10) What should this extension be called?

RESOLVED: The original name for this specification was NV_video_framebuffer because the motivation for creating this extension came from the need to expose a method for sending framebuffer objects to an SDI video output device. However, it has grown beyond that purpose and no longer even requires EXT_framebuffer_object to function. For these reasons, it has been renamed NV_present_video.

11) Should a "stacked fields" mode be added to allow the application to specify two fields vertically concatenated in one buffer?

RESOLVED: No. The stacked fields in previous extensions were a workaround to allow the application to specify two fields at once with an API that only accepted one image at a time. Since this extension requires all buffers that make up a frame to be specified simultaneously, stacked fields are not needed.

12) *Should there be a separate function for presenting output data for one stream?*

    RESOLVED: Yes. To clarify the different types of data needed for single and dual stream modes, two separate entry points are provided.

13) *Should we allow users to override the mode-defined mapping between frame-height buffer lines and field lines?*

    RESOLVED: No. Not only does this seem unnecessary, it is also impractical. If a mode has an odd number of lines, the application would need to specify incorrectly sized buffers to satisfy the line choosing rules as they are specified currently.

Revision History

Revision 6, 2008/2/20
    -Public specification

Name

    NV_primitive_restart

Name Strings

    GL_NV_primitive_restart

Notice

IP Status

    NVIDIA Proprietary.

Status

    Implemented in CineFX (NV30) Emulation driver, August 2002.
    Shipping in Release 40 NVIDIA driver for CineFX hardware, January 2003.

Version

    NVIDIA Date: August 29, 2002 (version 0.1)

Number

    285

Dependencies

    Written based on the wording of the OpenGL 1.3 specification.

Overview

This extension allows applications to easily and inexpensively restart a primitive in its middle. A "primitive restart" is simply the same as an End command, followed by another Begin command with the same mode as the original. The typical expected use of this feature is to draw a mesh with many triangle strips, though primitive restarts are legal for all primitive types, even for points (where they are not useful).

Although the EXT_multi_draw_arrays extension did reduce the overhead of such drawing techniques, they still remain more expensive than one would like.

This extension provides an extremely lightweight primitive restart, which is accomplished by allowing the application to choose a special index number that signals that a primitive restart should occur, rather than a vertex being provoked. This index can be an arbitrary 32-bit integer for maximum application convenience.

In addition, for full orthogonality, a special OpenGL command is provided to restart primitives when in immediate mode. This command is not likely to increase performance in any significant fashion, but providing it greatly simplifies the specification and implementation of display list compilation and indirect rendering.

Issues

* *What should the default primitive restart index be?*

RESOLVED: Zero. It's tough to pick another number that is meaningful for all three element data types. In practice, apps are likely to set it to 0xFFFF or 0xFFFFFFFF.

* *Are primitives other than triangle strips supported?*

RESOLVED: Yes. One example of how this can be useful is for rendering a heightfield. The "standard" way to render a heightfield uses a number of triangle strips, one for each row of the grid. Another method, which can produce higher-quality meshes, is to render a number of 8-triangle triangle fans. This has the effect of alternating the direction of tessellation, as shown in the diagram below. Primitive restarts enhance the performance of both techniques.

```
-------------------------        -------------------------
| /| /| /| /| /| /| /| /|        |\ | /|\ | /|\ | /|\ | /|
|/ |/ |/ |/ |/ |/ |/ |/ |        | \|/ | \|/ | \|/ | \|/ |
-------------------------        ---*-----*-----*-----*---
| /| /| /| /| /| /| /| /|        | /|\ | /|\ | /|\ | /|\ |
|/ |/ |/ |/ |/ |/ |/ |/ |        |/ | \|/ | \|/ | \|/ | \|
-------------------------        -------------------------

        Two strips             Four fans (centers marked '*')
```

* *How is this feature turned on and off?*

RESOLVED: Via a glEnable/DisableClientState setting. It is not possible to select a restart index that is guaranteed to be unused.

* *Is the immediate mode PrimitiveRestartNV needed?*

RESOLVED: Yes. It is difficult to make indirect rendering to work without it, and it is near impossible to make display lists work without it. It is a very clean way to resolve these issues.

* *How is indirect rendering handled?*

RESOLVED: Because of PrimitiveRestartNV, it works very easily. PrimitiveRestartNV has a wire protocol and therefore it can easily be inserted as needed. The server tracks the current Begin mode, relieving the client of this burden.

Note that in practice, we expect that this feature is essentially useless for indirect rendering.

* *How does this extension interact with NV_element_array and NV_vertex_array_range?*

RESOLVED: It doesn't, not even for performance. It should be fast on hardware that supports the feature with or without the use of element arrays, with or without vertex array range.

* *Does this extension affect ArrayElement and DrawArrays, or just DrawElements?*

RESOLVED: All of them. It applies to ArrayElement and to the rest as a consequence. It is likely not useful with any other than DrawElements, but nevertheless not prohibited.

* *In the case of ArrayElement, what happens if the restart index is used outside Begin/End?*

RESOLVED: Since this is defined as being equivalent to a call to PrimitiveRestartNV, and PrimitiveRestartNV is an INVALID_OPERATION when not inside Begin/End, this is just an error.

* *For DrawRangeElements/LockArrays purposes, must the restart index lie within the start/end range?*

RESOLVED: No, this would to some extent defeat the point if the restart index was, e.g., 0xFFFFFFFF. I don't believe any spec language is required here, since hitting this index does not cause a vertex to be dereferenced.

* *Should this state push/pop?*

RESOLVED: Yes, as vertex array client state.

New Procedures and Functions

```
void PrimitiveRestartNV(void);
void PrimitiveRestartIndexNV(uint index);
```

New Tokens

Accepted by the <array> parameter of EnableClientState and DisableClientState, by the <cap> parameter of IsEnabled, and by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

```
    PRIMITIVE_RESTART_NV                          0x8558
```

Accepted by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

```
    PRIMITIVE_RESTART_INDEX_NV                    0x8559
```

Additions to Chapter 2 of the OpenGL 1.3 Specification (OpenGL Operation)

Add a section 2.6.X "Primitive Restarts", immediately after section 2.6.2 "Polygon Edges" (page 19):

"2.6.X Primitive Restarts

An OpenGL primitive may be restarted with the command

```
  void PrimitiveRestartNV(void)
```

Between the execution of a Begin and its corresponding End, this command is equivalent to a call to End, followed by a call to Begin where the mode argument is the same mode as that used by the previous Begin. Outside the execution of a Begin and its corresponding End, this command generates the error INVALID_OPERATION."

Add PrimitiveRestartNV to the list of commands that are allowed between Begin and End in section 2.6.3 "GL Commands within Begin/End" (page 19).

Add to section 2.8 "Vertex Arrays", after the description of ArrayElement (page 24):

"Primitive restarting is enabled or disabled by calling EnableClientState or DisableClientState with parameter PRIMITIVE_RESTART_NV. The command

```
  void PrimitiveRestartIndexNV(uint index)
```

specifies the index of a vertex array element that is treated specially when primitive restarting is enabled. When ArrayElement is called between an execution of Begin and the corresponding execution of End, if i is equal to PRIMITIVE_RESTART_INDEX_NV, then no vertex data is derefererenced, and no current vertex state is modified. Instead, it is as if PrimitiveRestartNV had been called."

Replace the last paragraph of section 2.8 "Vertex Arrays" (page 28) with the following:

"If the number of supported texture units (the value of MAX_TEXTURE_UNITS) is k, then the client state required to implement vertex arrays consists of 7+k boolean values, 5+k memory pointers, 5+k integer stride values, 4+k symbolic constants representing array types, 3+k integers representing values per element, and an unsigned integer representing the restart index. In the initial state, the boolean values are each disabled, the memory pointers are each null, the strides are each zero, the array types are each FLOAT, the integers representing values per element are each four, and the restart index is zero."

Additions to Chapter 3 of the OpenGL 1.3 Specification (Rasterization)

None.

Additions to Chapter 4 of the OpenGL 1.3 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 1.3 Specification (Special Functions)

Add to the end of Section 5.4 "Display Lists":

"PrimitiveRestartIndexNV is not compiled into display lists, but is executed immediately."

Additions to Chapter 6 of the OpenGL 1.3 Specification (State and State Requests)

None.

GLX Protocol

One new GL command is added.

The following rendering command is sent to the server as part of a glXRender request:

```
    PrimitiveRestartNV
        2           4               rendering command length
        2           ????            rendering command opcode
```

Errors

The error INVALID_OPERATION is generated if PrimitiveRestartNV is called outside the execution of Begin and the corresponding execution of End.

The error INVALID_OPERATION is generated if PrimitiveRestartIndexNV is called between the execution of Begin and the corresponding execution of End.

New State

Get Value	Get Command	Type	Initial Value	Sec	Attrib
PRIMITIVE_RESTART_NV	IsEnabled	B	FALSE	2.8	vertex-array
PRIMITIVE_RESTART_INDEX_NV	GetIntegerv	Z+	0	2.8	vertex-array

Name

    NV_register_combiners

Name Strings

    GL_NV_register_combiners

Notice

IP Status

    NVIDIA Proprietary.

Status

    Shipping (version 1.6)

Version

    NVIDIA Date: February 1, 2007 (version 1.7)
    $Id: //sw/main/docs/OpenGL/specs/GL_NV_register_combiners.txt#56 $

Number

    191

Dependencies

    ARB_multitexture, assuming the value of MAX_TEXTURE_UNITS_ARB is
    at least 2.

    Written based on the wording of the OpenGL 1.2 specification with
    the ARB_multitexture appendix E.

    NV_texture_shader affects the definition of this extension.

    ARB_depth_texture and ARB_shadow -or- SGIX_depth_texture and
    SGIX_shadow affect the definition of this extension.

Overview

    NVIDIA's next-generation graphics processor and its derivative designs
    support an extremely configurable mechanism know as "register combiners"
    for computing fragment colors.

    The register combiner mechanism is a significant redesign of NVIDIA's
    original TNT combiner mechanism as introduced by NVIDIA's RIVA
    TNT graphics processor. Familiarity with the TNT combiners will
    help the reader appreciate the greatly enhanced register combiners
    functionality (see the NV_texture_env_combine4 OpenGL extension
    specification for this background). The register combiner mechanism
    has the following enhanced functionality:

The numeric range of combiner computations is from [-1,1] (instead of TNT's [0,1] numeric range),

The set of available combiner inputs is expanded to include the secondary color, fog color, fog factor, and a second combiner constant color (TNT's available combiner inputs consist of only zero, a single combiner constant color, the primary color, texture 0, texture 1, and, in the case of combiner 1, the result of combiner 0).

Each combiner variable input can be independently scaled and biased into several possible numeric ranges (TNT can only complement combiner inputs).

Each combiner stage computes three distinct outputs (instead TNT's single combiner output).

The output operations include support for computing dot products (TNT has no support for computing dot products).

After each output operation, there is a configurable scale and bias applied (TNT's combiner operations builds in a scale and/or bias into some of its combiner operations).

Each input variable for each combiner stage is fetched from any entry in a combiner register set. Moreover, the outputs of each combiner stage are written into the register set of the subsequent combiner stage (TNT could only use the result from combiner 0 as a possible input to combiner 1; TNT lacks the notion of an input/output register set).

The register combiner mechanism supports at least two general combiner stages and then a special final combiner stage appropriate for applying a color sum and fog computation (TNT provides two simpler combiner stages, and TNT's color sum and fog stages are hard-wired and not subsumed by the combiner mechanism as in register combiners).

The register combiners fit into the OpenGL pipeline as a rasterization processing stage operating in parallel to the traditional OpenGL texture environment, color sum, AND fog application. Enabling this extension bypasses OpenGL's existing texture environment, color sum, and fog application processing and instead use the register combiners. The combiner and texture environment state is orthogonal so modifying combiner state does not change the traditional OpenGL texture environment state and the texture environment state is ignored when combiners are enabled.

OpenGL application developers can use the register combiner mechanism for very sophisticated shading techniques. For example, an approximation of Blinn's bump mapping technique can be achieved with the combiner mechanism. Additionally, multi-pass shading models that require several passes with unextended OpenGL 1.2 functionality can be implemented in several fewer passes with register combiners.

Issues

Should we expose the full register combiners mechanism?

RESOLUTION: NO. We ignore small bits of NV10 hardware functionality. The texture LOD input is ignored. We also ignore the inverts on input to the EF product.

Do we provide full gets for all the combiner state?

RESOLUTION: YES.

Do we parameterize combiner input and output updates to avoid enumerant explosions?

RESOLUTION: YES. To update a combiner stage input variable, you need to specify the <stage>, <portion>, and <variable>. To update a combiner stage output operation, you need to specify the <stage> and <portion>. This does mean that we need to add special Get routines that are likewise parameterized. Hence, GetCombinerInputParameter*, GetCombinerOutputParameter*, and GetFinalCombinerInputParameter*.

Is the register combiner functionality a super-set of the TNT combiner functionality?

Yes, but only in the sense of being a computational super-set. All computations performed with the TNT combiners can be performed with the register combiners, but the sequence of operations necessary to configure an identical computational result can be quite different.

For example, the TNT combiners have an operation that includes a final complement operation. The register combiners can perform range mappings only on inputs, but not on outputs. The register combiners can mimic the TNT operation with a post-operation complement only by taking pains to complement on input any uses of the output in later combiner stages.

What this does mean is that NV10's hardware functionality will permit support for both the NV_register_combiners AND NV_texture_env_combine4 extensions.

Note the existance of an "speclit" input complement bit supported by NV10 (but not accessible through the NV_register_combiners extensions).

Should we say anything about the precision of the combiner computations?

RESOLUTION: NO. The spec is written as if the computations are done on floating point values ranging from -1.0 to 1.0 (clamping is specified where this range is exceeded). The fact that NV10 does the computations as 9-bit signed fixed point is not mentioned in the spec. This permits a future design to support more precision or use a floating pointing representation.

What should the initial combiner state be?

RESOLUTION: See tables NV_register_combiners.4 and NV_register_combiners.5. The default state has one general combiner stage active that modulates the incoming color with texture 0. The final combiner is setup initially to implement OpenGL 1.2's standard color sum and fog stages.

What should happen to the TEXTURE0_ARB and TEXTUER1_ARB inputs if one or both textures are disabled?

RESOLUTION: The value of these inputs is undefined.

What do the TEXTURE0_ARB and TEXTURE1_ARB inputs correspond to? Does the number correspond to the absolute texture unit number or is the number based on how many textures are enabled (ie, TEXTURE_ARB0 would correspond to the 2nd texture unit if the 2nd unit is enabled, but the 1st is disabled).

RESOLUTION: The absolute texture unit.

This should be a lot less confusing to the programmer than having the texture inputs switch textures if texture 0 is disabled.

Note that the proposed hardware actually determines the TEXTURE0 and TEXTURE1 input based on which texture is enabled. This means it is up to the ICD to properly update the combiner state when just one texture is enabled. Since we will already have to do this to track the standard OpenGL texture environment for ARB_multitexture, we can do it for this extension too.

Should the combiners state be PushAttrib/PopAttrib'ed along with the texture state?

RESOLUTION: YES.

Should we advertise the LOD fractional input to the combiners?

RESOLUTION: NO.

There will be a performance impact when two combiner stages are enabled versus just one stage. Should we mention that somewhere?

RESOLUTION: NO. (But it is worth mentioning in this issues section.)

Should the scale and bias for the CombinerOutputNV be indicated by enumerants or specified outright as floats?

RESOLUTION: ENUMERANTS. While some future combiners might support an arbitrary scale & bias specified as floats, NV10 just does the enumerated options.

Should a dot product be computed in parralel with the sum of products?

RESOLUTION: YES (changed for version 1.6). The hardware is capable of summing the two dot products.

Versions of this specification prior to version 1.6 documented that an INVALID_OPERATION should be generated if either <abDotProduct> or <cdDotProduct> is true and then <sumOutput> is not GL_DISCARD. However, this check was never added to the driver and some applications found the mode useful.

Does the GL_E_TIMES_F_NV token for the final combiner perform any mapping on E or F before the mapping? Is the multiply signed? Can the result be negative?

RESOLUTION: Input mappings and component usage is applied to E or F before their product is computed. Yes, the product is a signed multiplication. The result can be negative, but the two allowed final combiner input mapping modes (GL_UNSIGNED_IDENTITY_NV and GL_UNSIGNED_INVERT_NV) both effectively clamp their results to [0,1].

A negative value resulting from the "E times F" product with the GL_UNSIGNED_IDENTITY_NV mapping mode would be clamped to zero.

A negative value resulting from the "E times F" product with the GL_UNSIGNED_INVERT_NV mpaping mode would be clamped to zero but then one minus that clamped result (zero) would generate one.

New Procedures and Functions

```
    void CombinerParameterfvNV(GLenum pname,
                               const GLfloat *params);

    void CombinerParameterivNV(GLenum pname,
                               const GLint *params);

    void CombinerParameterfNV(GLenum pname,
                              GLfloat param);

    void CombinerParameteriNV(GLenum pname,
                              GLint param);

    void CombinerInputNV(GLenum stage,
                         GLenum portion,
                         GLenum variable,
                         GLenum input,
                         GLenum mapping,
                         GLenum componentUsage);

    void CombinerOutputNV(GLenum stage,
                          GLenum portion,
                          GLenum abOutput,
                          GLenum cdOutput,
                          GLenum sumOutput,
                          GLenum scale,
                          GLenum bias,
                          GLboolean abDotProduct,
                          GLboolean cdDotProduct,
                          GLboolean muxSum);
    void FinalCombinerInputNV(GLenum variable,
                              GLenum input,
                              GLenum mapping,
                              GLenum componentUsage);

    void GetCombinerInputParameterfvNV(GLenum stage,
                                       GLenum portion,
                                       GLenum variable,
                                       GLenum pname,
                                       GLfloat *params);

    void GetCombinerInputParameterivNV(GLenum stage,
                                       GLenum portion,
                                       GLenum variable,
                                       GLenum pname,
                                       GLint *params);

    void GetCombinerOutputParameterfvNV(GLenum stage,
                                        GLenum portion,
                                        GLenum pname,
                                        GLfloat *params);

    void GetCombinerOutputParameterivNV(GLenum stage,
                                        GLenum portion,
                                        GLenum pname,
                                        GLint *params);

    void GetFinalCombinerInputParameterfvNV(GLenum variable,
                                            GLenum pname,
                                            GLfloat *params);

    void GetFinalCombinerInputParameterivNV(GLenum variable,
                                            GLenum pname,
                                            GLint *params);
```

New Tokens

```
    Accepted by the <cap> parameter of Enable, Disable, and IsEnabled,
    and by the <pname> parameter of GetBooleanv, GetIntegerv,
    GetFloatv, and GetDoublev:

        REGISTER_COMBINERS_NV              0x8522

    Accepted by the <stage> parameter of CombinerInputNV,
    CombinerOutputNV, GetCombinerInputParameterfvNV,
    GetCombinerInputParameterivNV, GetCombinerOutputParameterfvNV,
    and GetCombinerOutputParameterivNV:

        COMBINER0_NV                       0x8550
        COMBINER1_NV                       0x8551
        COMBINER2_NV                       0x8552
        COMBINER3_NV                       0x8553
        COMBINER4_NV                       0x8554
        COMBINER5_NV                       0x8555
        COMBINER6_NV                       0x8556
        COMBINER7_NV                       0x8557
```

Accepted by the <variable> parameter of CombinerInputNV, GetCombinerInputParameterfvNV, and GetCombinerInputParameterivNV:

VARIABLE_A_NV	0x8523
VARIABLE_B_NV	0x8524
VARIABLE_C_NV	0x8525
VARIABLE_D_NV	0x8526

Accepted by the <variable> parameter of FinalCombinerInputNV, GetFinalCombinerInputParameterfvNV, and GetFinalCombinerInputParameterivNV:

VARIABLE_A_NV	
VARIABLE_B_NV	
VARIABLE_C_NV	
VARIABLE_D_NV	
VARIABLE_E_NV	0x8527
VARIABLE_F_NV	0x8528
VARIABLE_G_NV	0x8529

Accepted by the <input> parameter of CombinerInputNV:

ZERO		(not new)
CONSTANT_COLOR0_NV	0x852A	
CONSTANT_COLOR1_NV	0x852B	
FOG		(not new)
PRIMARY_COLOR_NV	0x852C	
SECONDARY_COLOR_NV	0x852D	
SPARE0_NV	0x852E	
SPARE1_NV	0x852F	
TEXTURE0_ARB		(see ARB_multitexture)
TEXTURE1_ARB		(see ARB_multitexture)

Accepted by the <mapping> parameter of CombinerInputNV:

UNSIGNED_IDENTITY_NV	0x8536
UNSIGNED_INVERT_NV	0x8537
EXPAND_NORMAL_NV	0x8538
EXPAND_NEGATE_NV	0x8539
HALF_BIAS_NORMAL_NV	0x853A
HALF_BIAS_NEGATE_NV	0x853B
SIGNED_IDENTITY_NV	0x853C
SIGNED_NEGATE_NV	0x853D

```
    Accepted by the <input> parameter of FinalCombinerInputNV:

        ZERO                                              (not new)
        CONSTANT_COLOR0_NV
        CONSTANT_COLOR1_NV
        FOG                                               (not new)
        PRIMARY_COLOR_NV
        SECONDARY_COLOR_NV
        SPARE0_NV
        SPARE1_NV
        TEXTURE0_ARB                                      (see ARB_multitexture)
        TEXTURE1_ARB                                      (see ARB_multitexture)
        E_TIMES_F_NV                           0x8531
        SPARE0_PLUS_SECONDARY_COLOR_NV         0x8532

    Accepted by the <mapping> parameter of FinalCombinerInputNV:

        UNSIGNED_IDENTITY_NV
        UNSIGNED_INVERT_NV

    Accepted by the <scale> parameter of CombinerOutputNV:

        NONE                                              (not new)
        SCALE_BY_TWO_NV                        0x853E
        SCALE_BY_FOUR_NV                       0x853F
        SCALE_BY_ONE_HALF_NV                   0x8540

    Accepted by the <bias> parameter of CombinerOutputNV:

        NONE                                              (not new)
        BIAS_BY_NEGATIVE_ONE_HALF_NV           0x8541

    Accepted by the <abOutput>, <cdOutput>, and <sumOutput> parameter
    of CombinerOutputNV:

        DISCARD_NV                             0x8530
        PRIMARY_COLOR_NV
        SECONDARY_COLOR_NV
        SPARE0_NV
        SPARE1_NV
        TEXTURE0_ARB                                      (see ARB_multitexture)
        TEXTURE1_ARB                                      (see ARB_multitexture)

    Accepted by the <pname> parameter of GetCombinerInputParameterfvNV
    and GetCombinerInputParameterivNV:

        COMBINER_INPUT_NV                      0x8542
        COMBINER_MAPPING_NV                    0x8543
        COMBINER_COMPONENT_USAGE_NV            0x8544
    Accepted by the <pname> parameter of GetCombinerOutputParameterfvNV
    and GetCombinerOutputParameterivNV:

        COMBINER_AB_DOT_PRODUCT_NV         0x8545
        COMBINER_CD_DOT_PRODUCT_NV         0x8546
        COMBINER_MUX_SUM_NV                0x8547
        COMBINER_SCALE_NV                  0x8548
        COMBINER_BIAS_NV                   0x8549
        COMBINER_AB_OUTPUT_NV              0x854A
        COMBINER_CD_OUTPUT_NV              0x854B
        COMBINER_SUM_OUTPUT_NV             0x854C

    Accepted by the <pname> parameter of CombinerParameterfvNV,
    CombinerParameterivNV, GetBooleanv, GetDoublev, GetFloatv, and
    GetIntegerv:

        CONSTANT_COLOR0_NV
        CONSTANT_COLOR1_NV

    Accepted by the <pname> parameter of CombinerParameterfvNV,
    CombinerParameterivNV, CombinerParameterfNV, CombinerParameteriNV,
    GetBooleanv, GetDoublev, GetFloatv, and GetIntegerv:

        NUM_GENERAL_COMBINERS_NV           0x854E
        COLOR_SUM_CLAMP_NV                 0x854F

    Accepted by the <pname> parameter of GetFinalCombinerInputParameterfvNV
    and GetFinalCombinerInputParameterivNV:

        COMBINER_INPUT_NV
        COMBINER_MAPPING_NV
        COMBINER_COMPONENT_USAGE_NV

    Accepted by the <pname> parameter of GetBooleanv, GetDoublev,
    GetFloatv, and GetIntegerv:

        MAX_GENERAL_COMBINERS_NV           0x854D
```

Additions to Chapter 2 of the 1.2 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.2 Specification (Rasterization)

-- Figure 3.1 "Rasterization" (page 58)

```
     +  Change the "Texturing" block to say "Texture Fetching".

     +  Insert a new block between "Texture Fetching" and "Color Sum".
        Name the new block "Texture Environment Application".

     +  Insert a new block after "Texture Fetching".  Name the new block
        "Register Combiners Application".

     +  The output of the "Texture Fetching" stage feeds to both "Texture
        Environment Application" and "Register Combiners Application".
```

+ The input for "Color Sum" comes from "Texture Environment Application".

+ The output to "Fragments" is switched (controlled by Disable/Enable REGISTER_COMBINERS_NV) between the output of "Fog" and "Register Combiners Application".

Essentially, when register combiners are enabled, the entire standard texture environment application, color sum, and fog blocks are replaced with the single register combiners block. [Note that this is different from how the NV_texture_env_combine4 extension works; that extension controls the texture environment application block, but still uses the standard color sum and fog blocks.]

-- NEW Section 3.8.12 "Register Combiners Application"

"In parallel to the texture application, color sum, and fog processes described in sections 3.8.10, 3.9, and 3.10, register combiners provide a means of computing fcoc, the final combiner output color, for each fragment generated by rasterization.

The register combiners consist of two or more general combiner stages arranged in a fixed sequence ordered by each combiner stage's number. An implementation supports a maximum number of general combiners stages, which may be queried by calling GetIntegerv with the symbolic constant MAX_GENERAL_COMBINERS_NV. Implementations must support at least two general combiner stages. The general combiner stages are named COMBINER0_NV, COMBINER1_NV, and so on.

Each general combiner in the sequence receives its inputs and computes its outputs in an identical manner. At the end of the sequence of general combiner stages, there is a final combiner stage that operates in a different manner than the general combiner stages. The general combiner operation is described first, followed by a description of the final combiner operation.

Each combiner stage (the general combiner stages and the final combiner stage) has an associated combiner register set. Each combiner register set contains <n> RGBA vectors with components ranging from -1.0 to 1.0 where <n> is 8 plus the maximum number of active textures supported (that is, the implementation's value for MAX_TEXTURE_UNITS_ARB). The combiner register set entries are listed in the table NV_register_combiners.1.

[Table NV_register_combiners.1]

Register Name	Initial Value	Reference	Output Status
ZERO	0	-	read only
CONSTANT_COLOR0_NV	ccc0	Section 3.8.12.1	read only
CONSTANT_COLOR1_NV	ccc1	Section 3.8.12.1	read only
FOG	Cf	Section 3.10	read only
PRIMARY_COLOR_NV	cpri	Section 2.13.1	read/write
SECONDARY_COLOR_NV	csec	Section 2.13.1	read/write
SPARE0_NV	see below	Section 3.8.12	read/write
SPARE1_NV	undefined	Section 3.8.12	read/write
TEXTURE0_ARB	CT0	Figure E.2	read/write
TEXTURE1_ARB	CT1	Figure E.2	read/write
TEXTURE<i>_ARB	CT<i>	Figure E.2	read/write

The register set of COMBINER0_NV, the first combiner stage,
is initialized as described in table NV_register_combiners.1.

The initial value of the alpha portion of register SECONDARY_COLOR_NV
is undefined. The initial value of the alpha portion of register
SPARE0_NV is the alpha component of texture 0 if texturing is
enabled for texture 0; however, the initial value of the RGB portion
SPARE0_NV is undefined. The initial value of the SPARE1_NV register
is undefined. The initial of registers TEXTURE0_ARB, TEXTURE1_ARB,
and TEXTURE<i>_ARB are undefined if texturing is not enabled for
textures 0, 1, and <i>, respectively.

The mapping of texture components to components of texture registers
is summarized in Table NV_register_combiners.2. In the following
table, At, Lt, It, Rt, Gt, Bt, and Dt, are the filtered texel
values.

[Table NV_register_combiners.2]: Correspondence of texture components to register components for texture registers.

Base Internal Format	RGB Values	Alpha Value
ALPHA	0, 0, 0	At
LUMINANCE	Lt, Lt, Lt	1
LUMINANCE_ALPHA	Lt, Lt, Lt	At
INTENSITY	It, It, It	It
RGB	Rt, Gt, Bt	1
RGBA	Rt, Gt, Bt	At
DEPTH_COMPONENT (when TEXTURE_COMPARE_MODE_ARB is NONE -or- TEXTURE_COMPARE_SGIX is false)	0, 0, 0,	Lt
DEPTH_COMPONENT (when TEXTURE_COMPARE_MODE_ARB is COMPARE_R_TO_TEXTURE -or- TEXTURE_COMPARE_SGIX is true)	Lt, Lt, Lt,	Lt
HILO_NV	0, 0, 0,	0
DSDT_NV	0, 0, 0,	0
DSDT_MAG_NV	0, 0, 0,	0
DSDT_MAG_INTENSITY_NV	0, 0, 0,	It

Note that the ALPHA, DEPTH_COMPONENT, and DSDT_MAG_INTENSITY_NV base internal formats are mapped to components differently than one could infer from the standard texture environment operations with this formats. A texture's DEPTH_TEXTURE_MODE_ARB state (see the ARB_depth_texture extension) is irrelevant for determining the correspondence of texture components to register components for texture registers when REGISTER_COMBINERS_NV is enabled.

3.8.12.1 Combiner Parameters

Combiner parameters are specified by

```
    CombinerParameterfvNV(GLenum pname, const GLfloat *params);
    CombinerParameterivNV(GLenum pname, const GLint *params);
    CombinerParameterfNV(GLenum pname, GLfloat param);
    CombinerParameteriNV(GLenum pname, GLint param);
```

<pname> is a symbolic constant indicating which parameter is to be set as described in the table NV_register_combiners.3:

[Table NV_register_combiners.3]

Parameter	Name	Number of values	Type
ccc0	CONSTANT_COLOR0_NV	4	color
ccc1	CONSTANT_COLOR1_NV	4	color
ngc	NUM_GENERAL_COMBINERS_NV	1	positive integer
csc	COLOR_SUM_CLAMP_NV	1	boolean

<params> is a pointer to a group of values to which to set the indicated parameter. <param> is simply the indicated parameter. The number of values pointed to depends on the parameter being set as shown in the table above. Color parameters specified with CombinerParameter*NV are converted to floating-point values (if specified as integers) as indicated by Table 2.6 for signed integers. The floating-point color values are then clamped to the range [0,1].

The values ccc0 and ccc1 named by CONSTANT_COLOR0_NV and CONSTANT_COLOR1_NV are constant colors available for inputs to the combiner stages. The value ngc named by NUM_GENERAL_COMBINERS_NV is a positive integer indicating how many general combiner stages are active, that is, how many general combiner stages a fragment should be processed by. Setting ngc to a value less than one or greater than the value of MAX_GENERAL_COMBINERS_NV generates an INVALID_VALUE error. The value csc named by COLOR_SUM_CLAMP_NV is a boolean described in section 3.8.12.3.

3.8.12.2 General Combiner Stage Operation

The command

```
    CombinerInputNV(GLenum stage,
                    GLenum portion,
                    GLenum variable,
                    GLenum input,
                    GLenum mapping,
                    GLenum componentUsage);
```

controls the assignment of all the general combiner input variables. For the RGB combiner portion, these are Argb, Brgb, Crgb, and Drgb; and for the combiner alpha portion, these are Aa, Ba, Ca, and Da. The <stage> parameter is a symbolic constant of the form COMBINER<i>_NV, indicating that general combiner stage <i> is to be updated. The constant COMBINER<i>_NV = COMBINER0_NV + <i> where <i> is in the range 0 to <k>-1 and <k> is the implementation dependent value of MAX_COMBINERS_NV. The <portion> parameter may be either RGB or ALPHA and determines whether the RGB color vector or alpha scalar portion of the specified combiner stage is updated. The <variable> parameter may be one of VARIABLE_A_NV, VARIABLE_B_NV, VARIABLE_C_NV, or VARIABLE_D_NV and determines which respective variable of the specified combiner stage and combiner stage portion is updated.

The <input>, <mapping>, and <componentUsage> parameters specify the assignment of a value for the input variable indicated by <stage>, <portion>, and <variable>. The <input> parameter may be one of the register names from table NV_register_combiners.1.

The <componentUsage> parameter may be one of RGB, ALPHA, or BLUE.

When the <portion> parameter is RGB, a <componentUsage> parameter of RGB indicates that the RGB portion of the indicated register should be assigned to the RGB portion of the combiner input variable, while an ALPHA <componentUsage> parameter indicates that the alpha portion of the indicated register should be replicated across the RGB portion of the combiner input variable.

When the <portion> parameter is ALPHA, the <componentUsage> parameter of ALPHA indicates that the alpha portion of the indicated register should be assigned to the alpha portion of the combiner input variable, while a BLUE <componentUsage> parameter indicates that the blue component of the indicated register should be assigned to the alpha portion of the combiner input variable.

When the <portion> parameter is ALPHA, a <componentUsage> parameter of RGB generates an INVALID_OPERATION error. When the <portion> parameter is RGB, a <componentUsage> parameter of BLUE generates an INVALID_OPERATION error.

When the <portion> parameter is ALPHA, an <input> parameter of FOG generates an INVALID_OPERATION error. The alpha component of the fog register is only available in the final combiner. The alpha component of the fog register is the fragment's fog factor when fog is enabled; otherwise, the alpha component of the fog register is one.

Before the value in the register named by <input> is assigned to the specified input variable, a range mapping is performed based on <mapping>. The mapping may be one of the tokens from the table NV_register_combiners.4.

[Table NV_register_combiners.4]

Mapping Name	Mapping Function
UNSIGNED_IDENTITY_NV	max(0.0, e)
UNSIGNED_INVERT_NV	1.0 - min(max(e, 0.0), 1.0)
EXPAND_NORMAL_NV	2.0 * max(0.0, e) - 1.0
EXPAND_NEGATE_NV	-2.0 * max(0.0, e) + 1.0
HALF_BIAS_NORMAL_NV	max(0.0, e) - 0.5
HALF_BIAS_NEGATE_NV	-max(0.0, e) + 0.5
SIGNED_IDENTITY_NV	e
SIGNED_NEGATE_NV	-e

Based on the <mapping> parameter, the mapping function in the table above is evaluated for each element <e> of the input vector before assigning the result to the specified input variable. Note that the mapping for the RGB and alpha portion of each input variable is distinct.

Each general combiner stage computes the following ten expressions based on the values assigned to the variables Argb, Brgb, Crgb, Drgb, Aa, Ba, Ca, and Da as determined by the combiner state set by CombinerInputNV.

["gcc" stands for general combiner computation.]

```
   gcc1rgb = [ Argb[r]*Brgb[r], Argb[g]*Brgb[g], Argb[b]*Brgb[b] ]

   gcc2rgb = [ Argb[r]*Brgb[r] + Argb[g]*Brgb[g] + Argb[b]*Brgb[b],
               Argb[r]*Brgb[r] + Argb[g]*Brgb[g] + Argb[b]*Brgb[b],
               Argb[r]*Brgb[r] + Argb[g]*Brgb[g] + Argb[b]*Brgb[b] ]

   gcc3rgb = [ Crgb[r]*Drgb[r], Crgb[g]*Drgb[g], Crgb[b]*Drgb[b] ]

   gcc4rgb = [ Crgb[r]*Drgb[r] + Crgb[g]*Drgb[g] + Crgb[b]*Drgb[b],
               Crgb[r]*Drgb[r] + Crgb[g]*Drgb[g] + Crgb[b]*Drgb[b],
               Crgb[r]*Drgb[r] + Crgb[g]*Drgb[g] + Crgb[b]*Drgb[b] ]

   gcc5rgb = gcc1rgb + gcc3rgb

   gcc6rgb = gcc1rgb or gcc3rgb               [see below]

   gcc1a   = Aa * Ba

   gcc2a   = Ca * Da

   gcc3a   = gcc1a + gcc2a

   gcc4a   = gcc1a or gcc2a                   [see below]
```

The computation of gcc6rgb and gcc4a involves a special "or" operation. This operation evaluates to the left-hand operand if the alpha component of the combiner's SPARE0_NV register is less than 0.5; otherwise, the operation evaluates to the right-hand operand.

```
    The command

        CombinerOutputNV(GLenum stage,
                         GLenum portion,
                         GLenum abOutput,
                         GLenum cdOutput,
                         GLenum sumOutput,
                         GLenum scale,
                         GLenum bias,
                         GLboolean abDotProduct,
                         GLboolean cdDotProduct,
                         GLboolean muxSum);

    controls the general combiner output operation including designating
    the register set locations where results of the general combiner's
    three computations are written.  The <stage> and <portion>
    parameters take the same values as the respective parameters for
    CombinerInputNV.

    If the <portion> parameter is ALPHA, specifying a non-FALSE value
    for either of the parameters <abDotProduct> or <cdDotProduct>,
    generates an INVALID_VALUE error.

    The <scale> parameter must be one of NONE, SCALE_BY_TWO_NV,
    SCALE_BY_FOUR_NV, or SCALE_BY_ONE_HALF_NV and specifies the
    value of the combiner stage's portion scale, either cscalergb or
    cscalea depending on the <portion> parameter, to 1.0, 2.0, 4.0,
    or 0.5, respectively.

    The <bias> parameter must be either NONE or
    BIAS_BY_NEGATIVE_ONE_HALF_NV and specifies the value of the
    combiner stage's portion bias, either cbiasrgb or cbiasa depending
    on the <portion> parameter, to 0.0 or -0.5, respectively.  If <scale>
    is either SCALE_BY_ONE_HALF_NV or SCALE_BY_FOUR_NV, a <bias> of
    BIAS_BY_NEGATIVE_ONE_HALF_NV generates an INVALID_OPERATION error.

    If the <abDotProduct> parameter is FALSE, then

       if <portion> is RGB,     out1rgb = max(min(gcc1rgb + cbiasrgb) * cscalergb, 1), -1)
       if <portion> is ALPHA,   out1a   = max(min((gcc1a + cbiasa) * cscalea, 1), -1)

    otherwise <portion> must be RGB and

       out1rgb = max(min((gcc2rgb + cbiasrgb) * cscalergb, 1), -1)

    If the <cdDotProduct> parameter is FALSE, then

       if <portion> is RGB,      out2rgb = max(min((gcc3rgb + cbiasrgb) * cscalergb, 1), -1)
       if <portion> is ALPHA,    out2a   = max(min((gcc2a + cbiasa) * cscalea, 1), -1)

    otherwise <portion> must be RGB so

       out2rgb = max(min((gcc4rgb + cbiasrgb) * cscalergb, 1), -1)
```

```
If the <muxSum> parameter is FALSE, then

   if <portion> is RGB,      out3rgb = max(min((gcc5rgb + cbiasrgb) * cscalergb, 1), -1)
   if <portion> is ALPHA,    out3a   = max(min((gcc3a + cbiasa) * cscalea, 1), -1)

otherwise

   if <portion> is RGB,      out3rgb = max(min((gcc6rgb + cbiasrgb) * cscalergb, 1), -1)
   if <portion> is ALPHA,    out3a   = max(min((gcc4a + cbiasa) * cscalea, 1), -1)
```

out1rgb, out2rgb, and out3rgb are written to the RGB portion of combiner stage's registers named by <abOutput>, <cdOutput>, and <sumOutput> respectively. out1a, out2a, and out3a are written to the alpha portion of combiner stage's registers named by <abOutput>, <cdOutput>, and <sumOutput> respectively. The parameters <abOutput>, <cdOutput>, and <sumOutput> must be either DISCARD_NV or one of the register names from table NV_register_combiners.1 that has an output status of read/write. If an output is set to DISCARD_NV, that output is not written to any register. The error INVALID_OPERATION is generated if <abOutput>, <cdOutput>, and <sumOutput> do not all name unique register names (though multiple outputs to DISCARD_NV are legal).

When the general combiner stage's register set is written based on the computed outputs, the updated register set is copied to the register set of the subsequent combiner stage in the combiner sequence. Copied undefined values are likewise undefined. The subsequent combiner stage following the last active general combiner stage, indicated by the general combiner stage's number being equal to ngc-1, in the sequence is the final combiner stage. In other words, the number of general combiner stages each fragment is transformed by is determined by the value of NUM_GENERAL_COMBINERS_NV.

3.8.12.3 Final Combiner Stage Operation

The final combiner stage operates differently from the general combiner stages. While a general combiner stage updates its register set and passes the register set to the next combiner stage, the final combiner outputs an RGBA color fcoc, the final combiner output color. The final combiner stage is capable of applying the standard OpenGL color sum and fog operations, but has the configurability to be used for other purposes.

The command

```
   FinalCombinerInputNV(GLenum variable,
                        GLenum input,
                        GLenum mapping,
                        GLenum componentUsage);
```

controls the assignment of all the final combiner input variables. The variables A, B, C, D, E, and F are RGB vectors. The variable G is an alpha scalar. The <variable> parameter may be one of VARIABLE_A_NV, VARIABLE_B_NV, VARIABLE_C_NV, VARIABLE_D_NV, VARIABLE_E_NV, VARIABLE_F_NV, and VARIABLE_G_NV, and determines which respective variable of the final combiner stage is updated.

The <input>, <mapping>, and <componentUsage> parameters specify the assignment of a value for the input variable indicated by <variable>.

The <input> parameter may be any one of the register names from table NV_register_combiners.1 or be one of two pseudo-register names, either E_TIMES_F_NV or SPARE0_PLUS_SECONDARY_COLOR_NV. The value of E_TIMES_F_NV is the product of the value of variable E times the value of variable F. The value of SPARE0_PLUS_SECONDARY_COLOR_NV is the value the SPARE0_NV register mapped using the UNSIGNED_IDENITY_NV input mapping plus the value of the SECONDARY_COLOR_NV register mapped using the UNSIGNED_IDENTITY_NV input mapping. If csc, the color sum clamp, is non-FALSE, the value of SPARE0_PLUS_SECONDARY_COLOR_NV is first clamped to the range [0,1]. The alpha component of E_TIMES_F_NV and SPARE0_PLUS_SECONDARY_COLOR_NV is always zero.

When <variable> is one of VARIABLE_E_NV, VARIABLE_F_NV, or VARIABLE_G_NV and <input> is either E_TIMES_F_NV or SPARE0_PLUS_SECONDARY_COLOR_NV, generate an INVALID_OPERATION error. When <variable> is VARIABLE_A_NV and <input> is SPARE0_PLUS_SECONDARY_COLOR_NV, generate an INVALID_OPERATION error.

The <componentUsage> parameter may be one of RGB, BLUE, ALPHA (with certain restrictions depending on the <variable> and <input> as described below).

When the <variable> parameter is not VARIABLE_G_NV, a <componentUsage> parameter of RGB indicates that the RGB portion of the indicated register should be assigned to the RGB portion of the combiner input variable, while an ALPHA <componentUsage> parameter indicates that the alpha portion of the indicated register should be replicated across the RGB portion of the combiner input variable.

When the <variable> parameter is VARIABLE_G_NV, a <componentUsage> parameter of ALPHA indicates that the alpha component of the indicated register should be assigned to the alpha portion of the G input variable, while a BLUE <componentUsage> parameter indicates that the blue component of the indicated register should be assigned to the alpha portion of the G input variable.

The INVALID_OPERATION error is generated when <componentUsage> is BLUE and <variable> is not VARIABLE_G_NV. The INVALID_OPERATION error is generated when <componentUsage> is RGB and <variable> is VARIABLE_G_NV.

The INVALID_OPERATION error is generated when both the <input> parameter is either E_TIMES_F_NV or SPARE0_PLUS_SECONDARY_COLOR_NV and the <componentUsage> parameter is ALPHA or BLUE.

Before the value in the register named by <input> is assigned to the specified input variable, a range mapping is performed based on <mapping>. The mapping may be either UNSIGNED_IDENTITY_NV or UNSIGNED_INVERT_NV and operates as specified in table NV_register_combiners.4.

The final combiner stage computes the following expression based on the values assigned to the variables A, B, C, D, E, F, and G as determined by the combiner state set by FinalCombinerInputNV

```
  fcoc = [ min(ab[r] + iac[r] + D[r], 1.0),
           min(ab[g] + iac[g] + D[g], 1.0),
           min(ab[b] + iac[b] + D[b], 1.0),
           G ]
```

where

```
  ab   = [ A[r]*B[r], A[g]*B[g], A[b]*B[b] ]
  iac  = [ (1.0 -A [r])*C[r], (1.0 - A[g])*C[g], (1.0 - A[b])*C[b] ]
```

3.8.12.4 Required State

The state required for the register combiners is a bit indicating whether register combiners are enabled or disabled, an integer indicating how many general combiners are active, a bit indicating whether or not the color sum clamp to 1 should be performed, two RGBA constant colors, <n> sets of general combiner stage state where <n> is the value of MAX_GENERAL_COMBINERS_NV, and the final combiner stage state. The per-stage general combiner state consists of the RGB input portion state and the alpha input portion state. Each portion (RGB and alpha) of the per-stage general combiner state consists of: four integers indicating the input register for the four variables A, B, C, and D; four integers to indicate each variable's range mapping; four bits to indicate whether to use the alpha component of the input for each variable; a bit indicating whether the AB dot product should be output; a bit indicating whether the CD dot product should be output; a bit indicating whether the sum or mux output should be output; two integers to maintain the output scale and bias enumerants; three integers to maintain the output register set names. The final combiner stage state consists of seven integers to indicate the input register for the seven variables A, B, C, D, E, F, and G; seven integers to indicate each variable's range mapping; and seven bits to indicate whether to use the alpha component of the input for each variable.

The general combiner per-stage state is initialized as described in table NV_register_combiners.5.

[Table NV_register_combiners.5]

Portion	Variable	Input	Component Usage	Mapping
RGB	A	PRIMARY_COLOR_NV	RGB	UNSIGNED_IDENTITY_NV
RGB	B	ZERO	RGB	UNSIGNED_INVERT_NV
RGB	C	ZERO	RGB	UNSIGNED_IDENTITY_NV
RGB	D	ZERO	RGB	UNSIGNED_IDENTITY_NV
alpha	A	PRIMARY_COLOR_NV	ALPHA	UNSIGNED_IDENTITY_NV
alpha	B	ZERO	ALPHA	UNSIGNED_INVERT_NV
alpha	C	ZERO	ALPHA	UNSIGNED_IDENTITY_NV
alpha	D	ZERO	ALPHA	UNSIGNED_IDENTITY_NV

The final combiner stage state is initialized as described in table NV_register_combiners.6.

[Table NV_register_combiners.6]

Variable	Input	Component Usage	Mapping
A	FOG	ALPHA	UNSIGNED_IDENTITY_NV
B	SPARE0_PLUS_SECONDARY_COLOR_NV	RGB	UNSIGNED_IDENTITY_NV
C	FOG	RGB	UNSIGNED_IDENTITY_NV
D	ZERO	RGB	UNSIGNED_IDENTITY_NV
E	ZERO	RGB	UNSIGNED_IDENTITY_NV
F	ZERO	RGB	UNSIGNED_IDENTITY_NV
G	SPARE0_NV	ALPHA	UNSIGNED_IDENTITY_NV"

-- NEW Section 3.8.11 "Antialiasing Application"

Insert the following paragraph BEFORE the section's first paragraph:

"Register combiners are enabled or disabled using the generic Enable and Disable commands, respectively, with the symbolic constant REGISTER_COMBINERS_NV. If the register combiners are enabled (and not in color index mode), the fragment's color value is replaced with fcoc, the final combiner output color, computed in section 3.8.12; otherwise, the fragment's color value is the result of the fog application in section 3.10."

Additions to Chapter 4 of the 1.2 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.2 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

-- Section 6.1.3 "Enumerated Queries"

Change the first two sentences (page 182) to say:

"Other commands exist to obtain state variables that are identified by a category (clip plane, light, material, combiners, etc.) as well as a symbolic constant. These are"

```
    Add to the bottom of the list of function prototypes (page 183):

      void GetCombinerInputParameterfvNV(GLenum stage, GLenum portion,
                                         GLenum variable,
                                         GLenum pname, const GLfloat *params);
      void GetCombinerInputParameterivNV(GLenum stage, GLenum portion,
                                         GLenum variable,
                                         GLenum pname, const GLint *params);
      void GetCombinerOutputParameterfvNV(GLenum stage, GLenum portion,
                                          GLenum pname, const GLfloat *params);
      void GetCombinerOutputParameterivNV(GLenum stage, GLenum portion,
                                          GLenum pname, GLint *params);
      void GetFinalCombinerInputParameterfvNV(GLenum variable, GLenum pname,
                                              const GLfloat *params);
      void GetFinalCombinerInputParameterivNV(GLenum variable, GLenum pname,
                                              const GLfloat *params);

    Add the following paragraph to the end of the section (page 184):

    "The GetCombinerInputParameterfvNV,
    GetCombinerInputParameterivNV, GetCombinerOutputParameterfvNV,
    and GetCombinerOutputParameterivNV parameter <stage> may be one of
    COMBINER0_NV, COMBINER1_NV, and so on, indicating which general
    combiner stage to query.  The GetCombinerInputParameterfvNV,
    GetCombinerInputParameterivNV, GetCombinerOutputParameterfvNV,
    and GetCombinerOutputParameterivNV parameter <portion> may be
    either RGB or ALPHA, indicating which portion of the general
    combiner stage to query.  The GetCombinerInputParameterfvNV
    and GetCombinerInputParameterivNV parameter <variable> may
    be one of VARIABLE_A_NV, VARIABLE_B_NV, VARIABLE_C_NV,
    or VARIABLE_D_NV, indicating which variable of the general
    combiner stage to query.  The GetFinalCombinerInputParameterfvNV
    and GetFinalCombinerInputParameterivNV parameter <variable> may be one
    of VARIABLE_A_NV, VARIABLE_B_NV, VARIABLE_C_NV, VARIABLE_D_NV,
    VARIABLE_E_NV, VARIABLE_F_NV, or VARIABLE_G_NV."
```

Additions to the GLX Specification

```
    None.
```

GLX Protocol

```
   Thirteen new GL commands are added.

   The following seven rendering commands are sent to the sever as part
   of a glXRender request:

       CombinerParameterfNV
           2           12              rendering command length
           2           4136            rendering command opcode
           4           ENUM            pname
           4           FLOAT32         param
```

```
CombinerParameterfvNV
    2           8+4*n           rendering command length
    2           4137            rendering command opcode
    4           ENUM            pname
                0x852A   n=4    GL_CONSANT_COLOR0_NV
                0x852B   n=4    GL_CONSANT_COLOR1_NV
                0x854E   n=1    GL_NUM_GENERAL_COMBINERS_NV
                0x854F   n=1    GL_COLOR_SUM_CLAMP_NV
                else     n=0
    4*n         LISTofFLOAT32   params

CombinerParameteriNV
    2           12              rendering command length
    2           4138            rendering command opcode
    4           ENUM            pname
    4           INT32           param

CombinerParameterivNV
    2           8+4*n           rendering command length
    2           4139            rendering command opcode
    4           ENUM            pname
                0x852A   n=4    GL_CONSANT_COLOR0_NV
                0x852B   n=4    GL_CONSANT_COLOR1_NV
                0x854E   n=1    GL_NUM_GENERAL_COMBINERS_NV
                0x854F   n=1    GL_COLOR_SUM_CLAMP_NV
                else     n=0
    4*n         LISTofINT32     params

CombinerInputNV
    2           28              rendering command length
    2           4140            rendering command opcode
    4           ENUM            stage
    4           ENUM            portion
    4           ENUM            variable
    4           ENUM            input
    4           ENUM            mapping
    4           ENUM            componentUsage

CombinerOutputNV
    2           36              rendering command length
    2           4141            rendering command opcode
    4           ENUM            stage
    4           ENUM            portion
    4           ENUM            abOutput
    4           ENUM            cdOutput
    4           ENUM            sumOutput
    4           ENUM            scale
    4           ENUM            bias
    1           BOOL            abDotProduct
    1           BOOL            cdDotProduct
    1           BOOL            muxSum
    1           BOOL            unused
```

```
        FinalCombinerInputNV
            2           20              rendering command length
            2           4142            rendering command opcode
            4           ENUM            variable
            4           ENUM            input
            4           ENUM            mapping
            4           ENUM            componentUsage

    The remaining six commands are non-rendering commands.  These commands
    are sent separately (i.e., not as part of a glXRender or glXRenderLarge
    request), using the glXVendorPrivateWithReply request:

        GetCombinerInputParameterfvNV
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           7               request length
            4           1270            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           ENUM            stage
            4           ENUM            portion
            4           ENUM            variable
            4           ENUM            pname
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           m               reply length, m = (n==1 ? 0 : n)
            4                           unused
            4           CARD32          unused

            if (n=1) this follows:

            4           FLOAT32         params
            12                          unused

            otherwise this follows:

            16                          unused
            n*4         LISTofFLOAT32   params
        GetCombinerInputParameterivNV
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           7               request length
            4           1271            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           ENUM            stage
            4           ENUM            portion
            4           ENUM            variable
            4           ENUM            pname
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           m               reply length, m = (n==1 ? 0 : n)
            4                           unused
            4           CARD32          unused

            if (n=1) this follows:

            4           INT32           params
            12                          unused

            otherwise this follows:

            16                          unused
            n*4         LISTofINT32     params

        GetCombinerOutputParameterfvNV
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           6               request length
            4           1272            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           ENUM            stage
            4           ENUM            portion
            4           ENUM            pname
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           m               reply length, m = (n==1 ? 0 : n)
            4                           unused
            4           CARD32          unused

            if (n=1) this follows:

            4           FLOAT32         params
            12                          unused

            otherwise this follows:

            16                          unused
            n*4         LISTofFLOAT32   params
        GetCombinerOutputParameterivNV
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           6               request length
            4           1273            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           ENUM            stage
            4           ENUM            portion
            4           ENUM            pname
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           m               reply length, m = (n==1 ? 0 : n)
            4                           unused
            4           CARD32          unused

            if (n=1) this follows:

            4           INT32           params
            12                          unused

            otherwise this follows:

            16                          unused
            n*4         LISTofINT32     params

        GetFinalCombinerInputParameterfvNV
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           5               request length
            4           1274            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           ENUM            variable
            4           ENUM            pname
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           m               reply length, m = (n==1 ? 0 : n)
            4                           unused
            4           CARD32          unused

            if (n=1) this follows:

            4           FLOAT32         params
            12                          unused

            otherwise this follows:

            16                          unused
            n*4         LISTofFLOAT32   params
        GetFinalCombinerInputParameterivNV
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           5               request length
            4           1275            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           ENUM            variable
            4           ENUM            pname
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           m               reply length, m = (n==1 ? 0 : n)
            4                           unused
            4           CARD32          unused

            if (n=1) this follows:

            4           INT32           params
            12                          unused

            otherwise this follows:

            16                          unused
            n*4         LISTofINT32     params
```

Dependencies on NV_texture_shader

If NV_texture_shader is not supported, references to HILO_NV, DSDT_NV, DSDT_MAG_NV, and DSDT_MAG_INTENSITY_NV base internal formats in this document are invalid and should be ignored.

Dependencies on ARB_depth_texture and ARB_shadow -or- SGIX_depth_texture and SGIX_shadow

If ARB_depth_texture and ARB_shadow -or- SGIX_depth_texture and SGIX_shadow are not supported, references to the DEPTH_COMPONENT base internal format in this document are invalid and should be ignored.

If ARB_depth_texture and ARB_shadow are not supported, references to the DEPTH_TEXTURE_MODE_ARB state in this document are invalid and should be ignored.

Errors

INVALID_VALUE is generated when CombinerParameterfvNV or CombinerParameterivNV is called with <pname> set to NUM_GENERAL_COMBINERS and the value pointed to by <params> is less than one or greater or equal to the value of MAX_GENERAL_COMBINERS_NV.

INVALID_OPERATION is generated when CombinerInputNV is called with a <componentUsage> parameter of RGB and a <portion> parameter of ALPHA.

```
    INVALID_OPERATION is generated when CombinerInputNV is called
    with a <componentUsage> parameter of BLUE and a <portion> parameter
    of RGB.

    INVALID_OPERATION is generated When CombinerInputNV is called with a
    <componentUsage> parameter of ALPHA and an <input> parameter of FOG.

    INVALID_VALUE is generated when CombinerOutputNV is called with
    a <portion> parameter of ALPHA, but a non-FALSE value for either
    of the parameters <abDotProduct> or <cdDotProduct>.

    INVALID_OPERATION is generated when CombinerOutputNV is called with
    a <scale> of either SCALE_BY_ONE_HALF_NV or SCALE_BY_FOUR_NV and
    a <bias> of BIAS_BY_NEGATIVE_ONE_HALF_NV.

    INVALID_OPERATION is generated when CombinerOutputNV is called such
    that <abOutput>, <cdOutput>, and <sumOutput> do not all name unique
    register names (though multiple outputs to DISCARD_NV are legal).

    INVALID_OPERATION is generated when FinalCombinerInputNV
    is called where <variable> is one of VARIABLE_E_NV,
    VARIABLE_F_NV, or VARIABLE_G_NV and <input> is E_TIMES_F_NV or
    SPARE0_PLUS_SECONDARY_COLOR_NV.

    INVALID_OPERATION is generated when FinalCombinerInputNV
    is called where <variable> is VARIABLE_A_NV and <input> is
    SPARE0_PLUS_SECONDARY_COLOR_NV.

    INVALID_OPERATION is generated when FinalCombinerInputNV is called
    with VARIABLE_G_NV for <variable> and RGB for <componentUsage>.

    INVALID_OPERATION is generated when FinalCombinerInputNV is called
    with a value other than VARIABLE_G_NV for <variable> and BLUE for
    <componentUsage>.

    INVALID_OPERATION is generated when FinalCombinerInputNV is
    called where the <input> parameter is either E_TIMES_F_NV or
    SPARE0_PLUS_SECONDARY_COLOR_NV and the <componentUsage> parameter
    is ALPHA.
```

New State

```
 --  (NEW table 6.29, after p217)
```

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
REGISTER_COMBINERS_NV	B	IsEnabled	False	register combiners enable	3.8.11	texture/enable
NUM_GENERAL_COMBINERS_NV	Z+	GetIntegerv	1	number of active combiner stages	3.8.12.1	texture
COLOR_SUM_CLAMP_NV	B	GetBooleanv	True	whether or not SPARE0_PLUS_ SECONDARY_ COLOR_NV clamps combiner stages	3.8.12.1	texture
CONSTANT_COLOR0_NV	C	GetFloatv	0,0,0,0	combiner constant color zero	3.8.12.1	texture
CONSTANT_COLOR1_NV	C	GetFloatv	0,0,0,0	combiner constant color one	3.8.12.1	texture
COMBINER_INPUT_NV	Z8x#x2x4	GetCombinerInputParameter*NV	see 3.8.12.4	combiner input variables	3.8.12.2	texture
COMBINER_COMPONENT_USAGE_NV	Z3x#x2x4	GetCombinerInputParameter*NV	see 3.8.12.4	use alpha for combiner input	3.8.12.2	texture
COMBINER_MAPPING_NV	Z8x#x2x4	GetCombinerInputParameter*NV	see 3.8.12.4	complement combiner input	3.8.12.2	texture
COMBINER_AB_DOT_PRODUCT_NV	Bx#x2	GetCombinerOutputParameter*NV	False	output AB dot product	3.8.12.3	texture
COMBINER_CD_DOT_PRODUCT_NV	Bx#x2	GetCombinerOutputParameter*NV	False	output CD dot product	3.8.12.3	texture
COMBINER_MUX_SUM_NV	Bx#x2	GetCombinerOutputParameter*NV	False	output mux sum	3.8.12.3	texture
COMBINER_SCALE_NV	Z2x#x2	GetCombinerOutputParameter*NV	NONE	output scale	3.8.12.3	texture
COMBINER_BIAS_NV	Z2x#x2	GetCombinerOutputParameter*NV	NONE	output bias	3.8.12.3	texture
COMBINER_AB_OUTPUT_NV	Z7x#x2	GetCombinerOutputParameter*NV	DISCARD_NV	AB output register	3.8.12.3	texture
COMBINER_CD_OUTPUT_NV	Z7x#x2	GetCombinerOutputParameter*NV	DISCARD_NV	CD output register	3.8.12.3	texture
COMBINER_SUM_OUTPUT_NV	Z7x#x2	GetCombinerOutputParameter*NV	SPARE0_NV	sum output register	3.8.12.3	texture
COMBINER_INPUT_NV	Z10x7	GetFinalCombinerInputParameter*NV	see 3.8.12.4	final combiner input	3.8.12.4	texture
COMBINER_MAPPING_NV	Z2x7	GetFinalCombinerInputParameter*NV	UNSIGNED_IDENTITY_NV	final combiner input mapping	3.8.12.4	texture
COMBINER_COMPONENT_USAGE_NV	Z2x7	GetFinalCombinerInputParameter*NV	see 3.8.12.4	use alpha for final combiner input mapping	3.8.12.4	texture

[where # is the value of MAX_GENERAL_COMBINERS_NV]

New Implementation Dependent State

(table 6.24, p214) add the following entry:

Get Value	Type	Get Command	Minimum Value	Description	Sec	Attribute
MAX_GENERAL_COMBINERS_NV	Z+	GetIntegerv	2	Maximum num of general combiner stages	3.8.12	-

NVIDIA Implementation Details

```
    The effective range of the RGB portion of the final combiner should
    be [0,4] if the color sum clamp is false.  Exercising  this range
    requires assigning SPARE0_PLUS_SECONDARY_COLOR_NV to the D variable
    and either B or C or both B and C.  In practice this is a very
    unlikely configuration.

    However due to a bug in the GeForce 256 and Quadro hardware, values
    generated above 2 in the RGB portion of the final combiner will be
    computed incorrectly.  GeForce2 GTS and subsequent NVIDIA GPUs have
```

fixed this bug.

The behavior of the SIGNED_NEGATE_NV mapping mode is undefined on GeForce3 GPUs (NV20) when used to map the initial value of a texture register corresponding to an enabled texture with a base internal format of GL_DEPTH_COMPONENT and a GL_TEXTURE_COMPARE_MODE_ARB mode of GL_COMPARE_R_TO_TEXTURE (or for SGIX_shadow, GL_TEXTURE_COMPARE_SGIX mode of true) mode when multiple enabled textures have different values for GL_TEXTURE_COMPARE_FUNC_ARB (or for SGIX_shadow, GL_TEXTURE_COMPARE_OPERATOR_SGIX). Values subsequently assigned to such registers and then mapped with SIGNED_NEGATIE_NV operate as expected. This issue does not affect GeForce4 Ti (NV25) and subsequent GPUs.

Revision History

April 4, 2000 - Document that alpha component of the FOG register should be zero when fog is disabled. The Release 4 NVIDIA drivers have a bug where this is not always true (though it often still is). The bug is fixed in the Release 5 NVIDIA drivers.

June 8, 2000 - The alpha component of the FOG register is not available for use until the final combiner. The specification previously incorrectly stated:

"INVALID_OPERATION is generated When CombinerInputNV is called with a <portion> parameter of ALPHA and an <input> parameter of FOG."

It is actually the <componentUsage> (not the <portion>) that should not be allowed to be ALPHA. The Release 4 NVIDIA drivers implemented the above incorrect error check. The Release 5 (and later) NVIDIA drivers (after June 8, 2000) have fixed this bug and correctly implement the error based on <componentUsage>.

The specification previously did not allow BLUE for the <componentUsage> of the G variable in the final combiner. This is now allowed in the Release 5 (and later) NVIDIA drivers (after June 8, 2000). The Release 4 NVIDIA drivers do not permit BLUE for the <componentUsage> of the G variable and generate an INVALID_OPERATION error if this is attempted. The Release 5 NVIDIA drivers (after June 8, 2000) have fixed this bug and permit BLUE for the <componentUsage> of the G variable.

August 11, 2000 - The "mux" operation was incorrectly documented in previous versions of this specification. The correct mux behave is as follows:

```
   spare0_alpha >= 0.5 ? C*D : A*B
```

or

```
   spare0_alpha <  0.5 ? A*B : C*D
```

Previous versions of this specification had the mux sense reversed.

October 31, 2000 - The initial general combiner state was misdocumented for the B variable. Previously, Table NV_register_combiners.5 said that the RGB and alpha inputs for B were GL_TEXTURE#_ARB and the RGB and alpha input mappings for B were GL_UNSIGNED_IDENTITY_NV. The table is now updated so that the RGB and alpha inputs for B are GL_ZERO and the RGB and alpha input mappings for B are GL_UNSIGNED_INVERT_NV. The implementation has always behaved in the manner described by the updated specification.

December 13, 2000 - Added a new table NV_register_combiners.2 describing the correspondence of texture components to register components for texture registers. This table is based on the table in the EXT_texture_env_combine extension. The table includes correspondences for HILO, DSDT, DSDT_MAG, DSDT_MAG_INTENSITY, and DEPTH_COMPONENT formatted textures when supported in conjunction with the NV_texture_shader, SGIX_depth_texture, and SGIX_shadow extensions.

Because a new table 2 was inserted, all the tables beyond it are renumbered.

Document the behavior of SIGNED_NEGATE_NV in conjunction with shadow mapping in the "NVIDIA Implementation Details" section.

June 28, 2002 - Properly document NV_register_combiners interactions with the ARB_depth_texture and ARB_shadow extensions (previously, the extension just addressed the SGIX versions of these extensions).

September 30, 2003 - Remove an error (not implemented in early NVIDIA drivers prior to Release 4x.xx drivers; implemented in Relase 4x.xx drivers; and again removed for Release 5x.xx drivers and up) that was meant to restrict the API to not allow the summing of dot product outputs. NVIDIA hardware handles this case correctly however so the functionality might as well be supported; some applications found it useful. The deleted error read:

> If the <abDotProduct> or <cdDotProduct> parameter is non-FALSE, the value of the <sumOutput> parameter must be GL_DISCARD_NV; otherwise, generate an INVALID_OPERATION error.

Name

NV_register_combiners2

Name Strings

GL_NV_register_combiners2

Notice

IP Status

NVIDIA Proprietary.

Status

Implemented.

Version

NVIDIA Date: February 11, 2004
Version 1.2

Number

227

Dependencies

Written based on the wording of the OpenGL 1.2.1 specification.

Assumes support for the NV_register_combiners extension (version 1.4).

Overview

The NV_register_combiners extension provides a powerful fragment coloring mechanism. This specification extends the register combiners functionality to support more color constant values that are unique for each general combiner stage.

The base register combiners functionality supports only two color constants. These two constants are available in every general combiner stage and in the final combiner.

When many general combiner stages are supported, more than two unique color constants is often required. The obvious way to extend the register combiners is to add several more color constant registers. But adding new unique color constant registers is expensive for hardware implementation because every color constant register must be available as an input to any stage.

In practice however, it is the total set of general combiner stages that requires more color constants, not each and every individual general combiner stage. Each individual general combiner stage typically requires only one or two color constants.

By keeping two color constant registers but making these two registers contain two unique color constant values for each general combiner stage, the hardware expense of supporting multiple color constants is minimized. Additionally, this scheme scales appropriately as more general combiner stages are added.

Issues

How do is compatibility maintained with the original register combiners?

RESOLUTION: Initially, per general combiner stage constants are disabled and the register combiners operate as described in the original NV_register_combiners specification. A distinct "per stage constants" enable exposes this extension's new functionality.

Where do the final combiner color constant values come from?

RESOLUTION: When "per stage constants" is enabled, the final combiner color constants continue to use the constant colors set with glCombinerParameterfvNV.

Is the alpha component of the SECONDARY_COLOR_NV register now initialized with the expected interpolated secondary color's alpha component?

RESOLUTION: Yes, see Revision History for details.

New Procedures and Functions

```
void CombinerStageParameterfvNV(GLenum stage,
                                GLenum pname,
                                const GLfloat *params);

void GetCombinerStageParameterfvNV(GLenum stage,
                                   GLenum pname,
                                   GLfloat *params);
```

New Tokens

Accepted by the <cap> parameter of Disable, Enable, and IsEnabled, and by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

```
    PER_STAGE_CONSTANTS_NV                        0x8535
```

Accepted by the <pname> parameter of CombinerStageParameterfvNV and GetCombinerStageParameterfvNV:

```
    CONSTANT_COLOR0_NV                         (see NV_register_combiners)
    CONSTANT_COLOR1_NV                         (see NV_register_combiners)
Accepted by the <stage> parameter of CombinerStageParameterfvNV and
GetCombinerStageParameterfvNV:

    COMBINER0_NV                                (see NV_register_combiners)
    COMBINER1_NV                                (see NV_register_combiners)
    COMBINER2_NV                                (see NV_register_combiners)
    COMBINER3_NV                                (see NV_register_combiners)
    COMBINER4_NV                                (see NV_register_combiners)
    COMBINER5_NV                                (see NV_register_combiners)
    COMBINER6_NV                                (see NV_register_combiners)
    COMBINER7_NV                                (see NV_register_combiners)
```

Additions to Chapter 2 of the 1.2 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.2 Specification (Rasterization)

-- Section 3.8.12 "Register Combiners Application"

Because the alpha component of the SECONDARY_COLOR_NV register is well-defined now (see Revision History) to be the alpha value of csec, STRIKE this sentence:

"The initial value of the alpha portion of register SECONDARY_COLOR_NV is undefined."

Add a paragraph immediately before section 3.8.12.1:

"The ccc0 and ccc1 values map to particular constant color values. The ccc0 and ccc1 mappings depend on whether per-stage constants are enabled or not. Per-stage constants are enabled and disabled with the Enable and Disable commands using the symbolic constant PER_STAGE_CONSTANTS_NV.

When per-stage constants are disabled, ccc0 and ccc1 are mapped to the register combiners' global color constant values, gccc0 and gccc1.

When per-stage constants are enabled, ccc0 and ccc1 depend on the combiner stage that inputs the COLOR_CONSTANT0_NV and COLOR_CONSTANT1_NV registers. Each general combiner stage # maps ccc0 and ccc1 to the per-stage values s#ccc0 and s#ccc1 respectively. The final combiner maps ccc0 and ccc1 to the values gccc0 and gccc1 (the same as if per-stage constants are disabled).

gccc0, gccc1, s#ccc0, and s#ccc1 are further described in the following section."

-- Section 3.8.12.1 "Combiner Parameters"

Change Table NV_register_combiners.3 to read "gccc0" instead of "ccc0" and "gccc1" instead of "ccc1".

    Change the first sentence of the last paragraph to read:

    "The values gccc0 and gccc1 named by CONSTANT_COLOR0_NV and
    CONSTANT_COLOR1_NV are global constant colors available for inputs to
    the final combiner stage and, when per-stage constants is disabled,
    to the general combiner stages."

    Add the following after the last paragraph in the section:

    "Per-stage combiner parameters are specified by

```
        void CombinerStageParameterfvNV(GLenum stage,
                                        GLenum pname,
                                        const GLfloat *params);
```

    The <stage> parameter is a symbolic constant of the form
    COMBINER<#>_NV, indicating the general combiner stage <#> whose
    parameter named by <pname> is to be updated. <pname> must be either
    CONSTANT_COLOR0_NV or CONSTANT_COLOR1_NV. <params> is a pointer
    to a group of four values to which to set the indicated parameter.
    The parameter names CONSTANT_COLOR0_NV and CONSTANT_COLOR1_NV
    update the per-stage color constants s#ccc0 and s#ccc1 respectively
    where # is the number of the specified general combiner stage.
    The floating-point color values are clamped to the range [0,1]
    when specified."

Additions to Chapter 4 of the 1.2 Specification (Per-Fragment Operations and the Frame Buffer)

    None

Additions to Chapter 5 of the 1.2 Specification (Special Functions)

    None

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

-- Section 6.1.3 "Enumerated Queries"

    Add to the bottom of the list of function prototypes (page 183):

```
        void GetCombinerStageParameterfvNV(GLenum stage,
                                           GLenum pname,
                                           GLfloat *params);
```

    Change the first sentence describing the register combiner queries
    to mention GetCombinerStageParameterfvNV so the sentence reads:

    "The GetCombinerInputParameterfvNV, GetCombinerInputParameterivNV,
    GetCombinerOutputParameterfvNV, GetCombinerOutputParameterivNV,
    and GetCombinerStageParameterfvNV parameter <stage> may be one of
    COMBINER0_NV, COMBINER1_NV, and so on, indicating which general
    combiner stage to query."

Additions to the GLX Specification

    None

GLX Protocol

    Two new GL commands are added.

    The following rendering command is sent to the sever as part of a
    glXRender request:

```
        CombinerParameterfvNV
            2           8+4*n           rendering command length
            2           ????            rendering command opcode
            4           ENUM            pname
                        0x852A   n=4    GL_CONSANT_COLOR0_NV
                        0x852B   n=4    GL_CONSANT_COLOR1_NV
                        else     n=0
            4*n         LISTofFLOAT32   params
```

    The remaining command is a non-rendering command. This commands
    is sent separately (i.e., not as part of a glXRender or glXRenderLarge
    request), using the glXVendorPrivateWithReply request:

```
        GetCombinerStageParameterfvNV
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           5               request length
            4           ????            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           ENUM            stage
            4           ENUM            pname
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           m               reply length, m = (n==1 ? 0 : n)
            4                           unused
            4           CARD32          unused

            if (n=1) this follows:

            4           FLOAT32         params
            12                          unused

            otherwise this follows:

            16                          unused
            n*4         LISTofFLOAT32   params
```

Errors

    None

New State

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
PER_STAGE_CONSTANTS_NV	B	IsEnabled	False	enable for	3.8.12	texture/enable
CONSTANT_COLOR0_NV	Cx#	GetCombinerStageParameterfvNV	0,0,0,0	per-stage constant color zero	3.8.12.1	texture
CONSTANT_COLOR1_NV	Cx#	GetCombinerStageParameterfvNV	0,0,0,0	per-stage constant color one	3.8.12.1	texture

[where # is the value of MAX_GENERAL_COMBINERS_NV]

New Implementation State

None

Revision History

Version 1.2 (February 11, 2004) - When describing the per-fragment register initialization within the combiners, the NV_register_combiners specification says "The initial value of the alpha portion of register SECONDARY_COLOR_NV is undefined." While this is true of NV1x GPUs, NV2x and beyond GPUs can properly initialize the alpha component of the SECONDARY_COLOR_NV register with the expected interpolated secondary color alpha. Unfortunately, due to a driver bug, the alpha components was always initialized to 1.0 in driver versions 56.90 (circa February 2004) and before. Drivers subsequent to 56.90 have this problem fixed. This specification is updated to indicate that SECONDARY_COLOR_NV initialization is well-defined and what you would expect now.

Version 1.1 (April 28, 2003) - The original specification failed to specify what should happen if a color component parameter for CombinerStageParameter*NV is outside the [0,1] range. Such values should be clamped to the [0,1] range.

NVIDIA drivers prior to May 2003 incorrectly failed to clamp color component values specified with CombinerStageParameter*NV to [0,1]. Instead, approximately "x-floor(x)" where x is a component value is used for rendering.

Name

    NV_texgen_emboss

Name Strings

    GL_NV_texgen_emboss

Notice

IP Status

NVIDIA Proprietary.

Status

Deprecated. Future NVIDIA drivers will NOT support this extension.
Developers are strongly encouraged to use NV_vertex_program instead
of this extension.

Version

```
NVIDIA Date: February 20, 2001
$Id: //sw/main/docs/OpenGL/specs/GL_NV_texgen_emboss.txt#22 $
```

Number

193

Dependencies

ARB_multitexture.

Written based on the wording of the OpenGL 1.2 specification and the
ARB_multitexture extension.

Overview

This extension provides a new texture coordinate generation mode
suitable for multitexture-based embossing (or bump mapping) effects.

Given two texture units, this extension generates the texture
coordinates of a second texture unit (an odd-numbered texture unit)
as a perturbation of a first texture unit (an even-numbered texture
unit one less than the second texture unit). The perturbation is
based on the normal, tangent, and light vectors. The normal vector
is supplied by glNormal; the light vector is supplied as a direction
vector to a specified OpenGL light's position; and the tanget
vector is supplied by the second texture unit's current texture
coordinate. The perturbation is also scaled by program-supplied
scaling constants.

If both texture units are bound to the same texture representing a
height field, by subtracting the difference between the resulting two
filtered texels, programs can achieve a per-pixel embossing effect.

Issues

Can you do embossing on any texture unit?

NO. Just odd numbered units. This meets a constraint of the proposed hardware implementation, and because embossing takes two texture units anyway, it shouldn't be a real limitation.

Can you just enable one coordinate of a texture unit for embossing?

Yes but NOT REALLY. The texture coordinate generation formula is specified such that only when ALL the coordinates are enabled and are using embossing, do you get the embossing computation. Otherwise, you get undefined values for texture coordinates enabled for texture coordinate generation and setup for embossing.

Does the light specified have to be enabled for embossing to work?

Yes, currently. But perhaps we could require implementations to enable a phantom light (the light colors would be black).

Could the emboss constant just be the reciprocal of the width and height of the texture units texture if that's what the programmer will have it be most of the time?

NO. Too much work and there may be reasons for the programmer to control this.

OpenGL's base texture environment functionality isn't powerful enough to do the subtraction needed for embossing. Where would you get powerful enough texture environment functionality.

Another extension. Try NV_register_combiners.

What is the interpretation of CT?

For the purposes of embossing, CT should be thought of as the vertex's tangent vector. This tangent vector indicates the direction on the "surface" where PCTs is not changing and PCTt is increasing.

Are the CT and PCT variables the user-supplied current texture coordinates?

YES. Except when the texture unit's texture coordinate evaluator is enabled, then CT and PCT use the respective evaluated texture coordinates.

This extension specification's language "Denote as CT the texture unit's current texture coordinates" and "Denote as PCT the previous texture unit's current texture coordinates" refers to the "current texture coordinates" OpenGL state which is the state specified via glTexCoord. Plus the exception for evaluators.

To be explicit, PCT is NOT the result of texgen or the texture matrix. Likewise, CT is NOT the result of texgen or the texture matrix. PCT and CT are the respective texture unit's evaluated texture coordinate if the vertex is evaluated with texture coordinate evaluation enabled, otherwise if the vertex is generated via vertex arrays with the respective texture coordinate array enabled, the texture coordinate from the texture coordinate array, otherwise the respective current texture coordinate is used.

New Procedures and Functions

None

New Tokens

Accepted by the <param> parameters of TexGend, TexGenf, and TexGeni when <pname> parameter is TEXTURE_GEN_MODE:

```
    EMBOSS_MAP_NV                     0x855F
```

When the <pname> parameter of TexGendv, TexGenfv, and TexGeniv is TEXTURE_GEN_MODE, then the array <params> may also contain EMBOSS_MAP_NV.

Accepted by the <pname> parameters of GetTexGendv, GetTexGenfv, GetTexGeniv, TexGend, TexGendv, TexGenf, TexGenfv, TexGeni, and TexGeniv:

```
    EMBOSS_LIGHT_NV                   0x855D
    EMBOSS_CONSTANT_NV                0x855E
```

Additions to Chapter 2 of the 1.2 Specification (OpenGL Operation)

-- Section 2.10.4 "Generating Texture Coordinates"

Change the last sentence in the 1st paragraph to:

"If <pname> is TEXTURE_GEN_MODE, then either <params> points to or <param> is an integer that is one of the symbolic constants OBJECT_LINEAR, EYE_LINEAR, SPHERE_MAP, or EMBOSS_MAP_NV."

Add these paragraphs after the 4th paragraph:

"When used with a suitable texture, suitable explicit texture coordinates, a suitable (extended) texture environment, suitable lighting parameters, and suitable embossing parameters, calling TexGen with TEXTURE_GEN_MODE indicating EMBOSS_MAP_NV can simulate the lighting effect of embossing on a polygon. The error INVALID_ENUM occurs when the active texture unit has an even number.

The emboss constant and emboss light parameters for controlling the EMBOSS_MAP_NV mode are specified by calling TexGen with pname set to EMBOSS_CONSTANT_NV and EMBOSS_LIGHT_NV respectively.

When pname is EMBOSS_CONSTANT_NV, param or what params points to is a scalar value. An error INVALID_ENUM occurs if pname is EMBOSS_CONSTANT_NV and coord is R or Q. An error INVALID_ENUM also occurs if pname is EMBOSS_CONSTANT_NV and the active texture unit number is even.

When pname is EMBOSS_LIGHT_NV, param or what params points to is a symbolic constant of the form LIGHTi, indicating that light i is to have the specified parameter set. An error INVALID_ENUM occurs if pname is EMBOSS_LIGHT_NV and coord is R or Q. An error INVALID_ENUM occurs if pname is EMBOSS_LIGHT_NV and the active texture unit number is even. An error INVALID_ENUM occurs if pname is EMBOSS_LIGHT_NV and the value i for LIGHTi is negative or is greater than or equal to the value of MAX_LIGHTS.

If TEXTURE_GEN_MODE indicates EMBOSS_MAP_NV, the generation function for the coordinates S, T, R, and Q is computed as follows.

Denote as L the light direction vector from the vertex's eye position to the position of the light specified by the coordinate's EMBOSS_LIGHT_NV state (the direction vector is computed as described in Section 3.13.1).

Denote as N the current normal after transformation to eye coordinates.

Denote as CT the texture unit's current texture coordinates transformed to eye coordinates by normal transformation (as described in Section 3.10.3) and normalized.

However, if the vertex is evaluated (as described in Section 5.1) and the texture unit's texture coordinate map is enabled, use the texture unit's evaluated texture coordinate to compute CT.

Denote as B the cross product of N and the <s,t,r> vector of CT.

```
  Bx = Ny*CTr - CTt*Nz
  By = Nz*CTs - CTr*Nx
  Bz = Nx*CTt - CTs*Ny
```

Denote as BN the normalized version of the vector B.

```
  BNx = Bx / sqrt(Bx*Bx + By*By + Bz*Bz);
  BNy = By / sqrt(Bx*Bx + By*By + Bz*Bz);
  BNz = Bz / sqrt(Bx*Bx + By*By + Bz*Bz);
```

Denote as T the cross product of B and N.

```
  Tx = BNy*Nz - Ny*BNz
  Ty = BNz*Nx - Nz*BNx
  Tz = BNx*Ny - Nx*BNy
```

Observe that BN and T are orthonormal.

Denote as PCT the previous texture unit's current texture coordinates. If the number of the texture unit for the texture coordinates being generated is n, then the previous texture unit is texture unit number n-1. Note that n is restricted to be odd.

However, if the vertex is evaluated (as described in Section 5.1) and the previous texture unit's texture coordinate map is enabled,

```
    use the previous texture unit's evaluated texture coordinate to
    compute PCT.

    Denote Ks as the S coordinate's EMBOSS_CONSTANT_NV state.  Denote Kt
    as the T coordinate's EMBOSS_CONSTANT_NV state.  These constants
    should typically be set to the reciprocal of the width and height
    respectively of the texture map used for embossing.

    Denote E as follows:

      Es = PCTs + Ks * (Lx*BNx + Ly*BNy + Lz*BNz) * PCTq
      Et = PCTt - Kt * (Lx*Tx + Ly*Ty + Lz*Tz) * PCTq
      Er = PCTr
      Eq = PCTq

    Then the value assigned to an s, t, r, and q coordinates are Es,
    Et, Er, and Eq respectively.  However, for this assignment to
    occur, the following three conditions must be met.  First, all the
    texture coordinate generation modes of all the texture coordinates
    (S, T, R, and Q) of the texture unit must be set to EMBOSS_MAP_NV.
    Second, all the texture coordinate generation modes of the texture
    unit must be enabled.  Third, the EMBOSS_LIGHT_NV parameters of
    coordinates S and T must be identical and the light and lighting
    must be enabled.  If these conditions are not met, the values of
    all coordinates in the texture unit with the EMBOSS_MAP_NV mode
    are undefined."

    The last paragraph's first sentence should be changed to:

    "The state required for texture coordinate generation comprises
    a five-valued integer for each coordinate indicating coordinate
    generation mode, and a bit for each coordinate to indicate whether
    texture coordinate generation is enabled or disabled.  In addition,
    four coefficients are required for the four coordinates for each
    of EYE_LINEAR and OBJECT_LINEAR; also, an emboss constant and
    emboss light are required for each of the four coordinates....
    The initial values for emboss constants and emboss lights are 1.0
    and LIGHT0 respectively."
```

Additions to Chapter 3 of the 1.2 Specification (Rasterization)

None

Additions to Chapter 4 of the 1.2 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.2 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

None

Additions to the GLX Specification

None

Errors

INVALID_ENUM is generated when TexGen is called with a <pname> of TEXTURE_GEN_MODE, a <param> value or value of what <params> points to of EMBOSS_MAP_NV, and the active texture unit is even.

INVALID_ENUM is generated when TexGen is called with a <pname> of EMBOSS_CONSTANT_NV and the active texture unit is even.

INVALID_ENUM is generated when TexGen is called with a <pname> of EMBOSS_LIGHT_NV and the active texture unit is even.

INVALID_ENUM is generated when TexGen is called with a <coord> of R or Q when <pname> indicates EMBOSS_CONSTANT_NV.

INVALID_ENUM is generated when TexGen is called with a <coord> of R or Q when <pname> indicates EMBOSS_LIGHT_NV.

INVALID_ENUM is generated when TexGen is called with a <pname> of EMBOSS_LIGHT_NV and the value of i for the parameter LIGHTi is negative or is greater than or equal to the value of MAX_LIGHTS.

New State

(table 6.14, p204) change the entry for TEXTURE_GEN_MODE to:

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
TEXTURE_GEN_MODE	4xZ5	GetTexGeniv	EYE_LINEAR	Function used for texgen (for s,t,r, and q)	2.10.4	texture
EMBOSS_CONSTANT_NV	4xR	GetTexGenfv	1.0	Scaling constant for emboss texgen	2.10.4	texture
EMBOSS_LIGHT_NV	4xZ8*	GetTexGeniv	LIGHT0	Light used for embossing.	2.10.4	texture

When ARB_multitexture is supported, the Type column is per-texture unit.

(the TEXTURE_GEN_MODE type changes from 4xZ3 to 4xZ5)

New Implementation State

None

Revision History

2001/02/20 - Status changed to deprecated.

Name

NV_texgen_reflection

Name Strings

GL_NV_texgen_reflection

Notice

Status

Shipping (version 1.0)
NVIDIA, Mesa 3.1, and ATI support this.

This extension's texture coordinate generation functionality is incoporated into the ARB_texture_cube_map extension. The same enumerant values are used.

The ARB_texture_cube_map functionality, including this texgen reflection functionality, is part of OpenGL 1.3 and subsequent revisions of the core OpenGL standard.

Version

June 17, 2003
$Date: 2003/06/17 $ $Revision: #10 $

Number

179

Dependencies

Written based on the wording of the OpenGL 1.2 specification but not dependent on it.

Overview

This extension provides two new texture coordinate generation modes that are useful texture-based lighting and environment mapping. The reflection map mode generates texture coordinates (s,t,r) matching the vertex's eye-space reflection vector. The reflection map mode is useful for environment mapping without the singularity inherent in sphere mapping. The normal map mode generates texture coordinates (s,t,r) matching the vertex's transformed eye-space normal. The normal map mode is useful for sophisticated cube map texturing-based diffuse lighting models.

Issues

Should we place the normal/reflection vector in the (s,t,r) texture coordinates or (s,t,q) coordinates?

RESOLUTION: (s,t,r). Even if the proposed hardware uses "q" for the third component, the API should claim to support generation of (s,t,r) and let the texture matrix (through a concatenation with the user-supplied texture matrix) move "r" into "q".

Should you be able to have some texture coordinates computing REFLECTION_MAP_NV and others not? Same question with NORMAL_MAP_NV.

RESOLUTION: YES. This is the way that SPHERE_MAP works. It is not clear that this would ever be useful though.

Should something special be said about the handling of the q texture coordinate for this spec?

RESOLUTION: NO. But the following paragraph is useful for implementors concerned about the handling of q.

The REFLECTION_MAP_NV and NORMAL_MAP_NV modes are intended to supply reflection and normal vectors for cube map texturing hardware. When these modes are used for cube map texturing, the generated texture coordinates can be thought of as a reflection vector. The value of the q texture coordinate then simply scales the vector but does not change its direction. Because only the vector direction (not the vector magnitude) matters for cube map texturing, implementations are free to leave q undefined when any of the s, t, or r texture coordinates are generated using REFLECTION_MAP_NV or NORMAL_MAP_NV.

New Procedures and Functions

None

New Tokens

Accepted by the <param> parameters of TexGend, TexGenf, and TexGeni when <pname> parameter is TEXTURE_GEN_MODE:

```
NORMAL_MAP_NV                     0x8511
REFLECTION_MAP_NV                 0x8512
```

When the <pname> parameter of TexGendv, TexGenfv, and TexGeniv is TEXTURE_GEN_MODE, then the array <params> may also contain NORMAL_MAP_NV or REFLECTION_MAP_NV.

Additions to Chapter 2 of the 1.2 Specification (OpenGL Operation)

-- Section 2.10.4 "Generating Texture Coordinates"

Change the last sentence in the 1st paragraph to:

"If <pname> is TEXTURE_GEN_MODE, then either <params> points to or <param> is an integer that is one of the symbolic constants

OBJECT_LINEAR, EYE_LINEAR, SPHERE_MAP, REFLECTION_MAP_NV, or NORMAL_MAP_NV."

Add these paragraphs after the 4th paragraph:

"If TEXTURE_GEN_MODE indicates REFLECTION_MAP_NV, compute the reflection vector r as described for the SPHERE_MAP mode. Then the value assigned to an s coordinate (the first TexGen argument value is S) is s = rx; the value assigned to a t coordinate is t = ry; and the value assigned to a r coordinate is r = rz. Calling TexGen with a <coord> of Q when <pname> indicates REFLECTION_MAP_NV generates the error INVALID_ENUM.

If TEXTURE_GEN_MODE indicates NORMAL_MAP_NV, compute the normal vector n' as described in section 2.10.3. Then the value assigned to an s coordinate (the first TexGen argument value is S) is s = nfx; the value assigned to a t coordinate is t = nfy; and the value assigned to a r coordinate is r = nfz. (The values nfx, nfy, and nfz are the components of nf.) Calling TexGen with a <coord> of Q when <pname> indicates REFLECTION_MAP_NV generates the error INVALID_ENUM.

The last paragraph's first sentence should be changed to:

"The state required for texture coordinate generation comprises a five-valued integer for each coordinate indicating coordinate generation mode, ..."

Additions to Chapter 3 of the 1.2 Specification (Rasterization)

None

Additions to Chapter 4 of the 1.2 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.2 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

None

Additions to the GLX Specification

None

Errors

INVALID_ENUM is generated when TexGen is called with a <coord> of Q when <pname> indicates REFLECTION_MAP_NV or NORMAL_MAP_NV.

New State

(table 6.14, p204) change the entry for TEXTURE_GEN_MODE to:

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
TEXTURE_GEN_MODE	4xZ5	GetTexGeniv	EYE_LINEAR	Function used for texgen (for s,t,r, and q)	2.10.4	texture

(the type changes from 4xZ3 to 4xZ5)

New Implementation State

None

Revision History

None

Name

    NV_texture_compression_vtc

Name Strings

    GL_NV_texture_compression_vtc

Notice

IP Status

    NVIDIA Proprietary.

Version

    NVIDIA Date: April 20, 2004
    $Id: //sw/main/docs/OpenGL/specs/GL_NV_texture_compression_vtc.txt#3 $

Number

    228

Dependencies

Written based on the wording of the OpenGL 1.2.1 specification.

ARB_texture_compression is required.

EXT_texture_compression_s3tc is required.

Overview

This extension adds support for the VTC 3D texture compression formats, which are analogous to the S3TC texture compression formats, with the addition of some retiling in the Z direction. VTC has the same compression ratio as S3TC and uses 4x4x1, 4x4x2, or 4x4x4 blocks.

Issues

* *Should the enumerants' (1) values and (2) names be reused from the S3TC extension?*

  RESOLVED: Yes and yes. There is such a close correspondence between the formats that introducing new values or names would serve no purpose.

* *Should the block alignment restrictions differ in any way from the block alignment restrictions in the S3TC extension?*

  RESOLVED: No, except for the addition of the Z-direction block alignment restriction for CompressedTexSubImage3D, which is analogous to the X and Y restrictions.

New Procedures and Functions

None.

New Tokens

Accepted by the <internalformat> parameter of TexImage3D and CompressedTexImage3DARB and the <format> parameter of CompressedTexSubImage2DARB:

```
    COMPRESSED_RGB_S3TC_DXT1_EXT                      0x83F0
    COMPRESSED_RGBA_S3TC_DXT1_EXT                     0x83F1
    COMPRESSED_RGBA_S3TC_DXT3_EXT                     0x83F2
    COMPRESSED_RGBA_S3TC_DXT5_EXT                     0x83F3
```

Additions to Chapter 2 of the OpenGL 1.2.1 Specification (OpenGL Operation)

None.

Additions to Chapter 3 of the OpenGL 1.2.1 Specification (Rasterization)

Modify the paragraph added to the end of the TexSubImage discussion (page 123) by EXT_texture_compression_s3tc to say:

"If the internal format of the texture image being modified is COMPRESSED_RGB_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT3_EXT, or COMPRESSED_RGBA_S3TC_DXT5_EXT, the texture is stored using one of several S3TC or VTC compressed texture image formats. Such images are easily edited along 4x4 texel boundaries, so the limitations on TexSubImage2D, TexSubImage3D, CopyTexSubImage2D, and CopyTexSubImage3D parameters are relaxed. These commands will result in an INVALID_OPERATION error only if one of the following conditions occurs:

* <width> is not a multiple of four or equal to TEXTURE_WIDTH.
* <height> is not a multiple of four or equal to TEXTURE_HEIGHT.
* <xoffset> or <yoffset> is not a multiple of four."

Modify the paragraph added to Section 3.8.2 "Alternate Image Specification" at the end of the CompressedTexImage section by EXT_texture_compression_s3tc to say:

"If <internalformat> is COMPRESSED_RGB_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT3_EXT, or COMPRESSED_RGBA_S3TC_DXT5_EXT, the compressed texture is stored using one of several S3TC or VTC compressed texture image formats. The S3TC texture compression algorithm supports only 2D images without borders, while the VTC texture compression algorithm supports only 3D images without borders. CompressedTexImage1DARB produces an INVALID_ENUM error if <internalformat> is an S3TC/VTC format. CompressedTexImage2DARB and CompressedTexImage3DARB will produce an INVALID_OPERATION error if <border> is non-zero."

Modify the paragraph added to Section 3.8.2 "Alternate Image Specification" at the end of the CompressedTexSubImage section by EXT_texture_compression_s3tc to say:

    "If the internal format of the texture image being modified is
    COMPRESSED_RGB_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT1_EXT,
    COMPRESSED_RGBA_S3TC_DXT3_EXT, or COMPRESSED_RGBA_S3TC_DXT5_EXT, the
    texture is stored using one of several S3TC or VTC compressed texture
    image formats. Since these algorithms support only 2D and 3D images,
    CompressedTexSubImage1DARB produces an INVALID_ENUM error if <format>
    is an S3TC/VTC format. Since S3TC/VTC images are easily edited along
    4x4 and 4x4x4 texel boundaries, the limitations on
    CompressedTexSubImage2D and CompressedTexSubImage3D are relaxed.
    CompressedTexSubImage2D and CompressedTexSubImage3D will result in an
    INVALID_OPERATION error only if one of the following conditions
    occurs:

* <width> is not a multiple of four or equal to TEXTURE_WIDTH.
* <height> is not a multiple of four or equal to TEXTURE_HEIGHT.
* <depth> is not a multiple of four or equal to TEXTURE_DEPTH.
* <xoffset>, <yoffset>, or <zoffset> is not a multiple of four."

Additions to Chapter 4 of the OpenGL 1.2.1 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 1.2.1 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 1.2.1 Specification (State and State Requests)

None.

Additions to Appendix A of the OpenGL 1.2.1 Specification (Invariance)

None.

GLX Protocol

None.

Errors

The INVALID_ENUM error that was generated by CompressedTexImage3DARB
if <internalformat> is COMPRESSED_RGB_S3TC_DXT1_EXT,
COMPRESSED_RGBA_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT3_EXT, or
COMPRESSED_RGBA_S3TC_DXT5_EXT no longer occurs.

INVALID_OPERATION is generated by CompressedTexImage3DARB if
<internalformat> is COMPRESSED_RGB_S3TC_DXT1_EXT,
COMPRESSED_RGBA_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT3_EXT, or
COMPRESSED_RGBA_S3TC_DXT5_EXT and <border> is not equal to zero.

The INVALID_ENUM error that was generated by
CompressedTexSubImage3DARB if <format> is
COMPRESSED_RGB_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT1_EXT,
COMPRESSED_RGBA_S3TC_DXT3_EXT, or COMPRESSED_RGBA_S3TC_DXT5_EXT no
longer occurs.

INVALID_OPERATION is generated by TexSubImage3D or CopyTexSubImage3D if INTERNAL_FORMAT is COMPRESSED_RGB_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT3_EXT, or COMPRESSED_RGBA_S3TC_DXT5_EXT and any of the following apply: <width> is not a multiple of four or equal to TEXTURE_WIDTH; <height> is not a multiple of four or equal to TEXTURE_HEIGHT; <xoffset> or <yoffset> is not a multiple of four.

INVALID_OPERATION is generated by CompressedTexSubImage3D if INTERNAL_FORMAT is COMPRESSED_RGB_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT3_EXT, or COMPRESSED_RGBA_S3TC_DXT5_EXT and any of the following apply: <width> is not a multiple of four or equal to TEXTURE_WIDTH; <height> is not a multiple of four or equal to TEXTURE_HEIGHT; <depth> is not a multiple of four or equal to TEXTURE_DEPTH; <xoffset> <yoffset>, or <zoffset> is not a multiple of four.

See also errors in the GL_ARB_texture_compression and GL_EXT_texture_compression_s3tc specifications.

New State

None.

Appendix

VTC Compressed Texture Image Formats

Each VTC compression format is similar to a corresponding S3TC compression format, but where an S3TC block encodes a 4x4 block of texels, a VTC block encodes a 4x4x1, 4x4x2, or 4x4x4 block of texels. If the depth of the image is four or greater, 4x4x4 blocks are used, and if the depth is 1 or 2, 4x4x1 or 4x4x2 blocks are used.

The size in bytes of a VTC image with dimensions w, h, and d is:

```
    ceil(w/4) * ceil(h/4) * d * blocksize,
```

where blocksize is the size of an analogous 4x4 S3TC block and is either 8 or 16 bytes.

The block containing a texel at location (x,y,z) starts at an offset inside the image of:

```
    blocksize * min(d,4) * (floor(x/4) +
                            ceil(w/4) * (floor(y/4) +
                                         ceil(h/4) * floor(z/4)))
```

bytes.

A 4x4x1 block of each of the four formats is stored in exactly the same way that a 4x4 block of the analogous S3TC format is stored.

A 4x4x2 or 4x4x4 block is stored as two or four consecutive 4x4 blocks of the analogous S3TC format, one for each layer inside the block. For example, a 4x4x2 DXT1 block consists of 16 bytes in total. The first 8 bytes encode the texels at locations (0,0,0) through (3,3,0), and the second 8 bytes encode the texels at locations (0,0,1) through (3,3,1).

For definitions of the S3TC formats, please refer to the EXT_texture_compression_s3tc specification.

Revision History

April 20, 2004 - Relax restrictions on depth and zoffset for CopyTexSubImage3D and TexSubImage3D commands. Previous restrictions required 1) the image level's depth to be 1 for CopyTexSubImage3D to work (making the command useless in practice) and 2) the depth and zoffset for TexSubImage3D to be a multiple 4. If these restrictions were violated, an INVALID_OPERATION error was documented to be generated. NVIDIA Release 60 drivers after April 20, 2004 relax these restrictions. Note the restrictions on CompressedTexSubImage3D that depth and zoffset must be multiples of 4 still exist because the VTC block is a 3D 4x4x4 block (or 4x4x2 and 4x4x1 in the end cases).

Name

    NV_texture_env_combine4

Name Strings

    GL_NV_texture_env_combine4

Notice

IP Status

    NVIDIA Proprietary.

Version

```
    NVIDIA Date: January 18, 2001
    $Date: 1999/06/21 13:54:17 $ $Revision: 1.2 $
    $Id: //sw/main/docs/OpenGL/specs/GL_NV_texture_env_combine4.txt#13 $
```

Number

    195

Dependencies

    EXT_texture_env_combine is required and is modified by this extension
    ARB_multitexture affects the definition of this extension

Overview

```
    New texture environment function COMBINE4_NV allows programmable
    texture combiner operations, including

        ADD                     Arg0 * Arg1 + Arg2 * Arg3
        ADD_SIGNED_EXT          Arg0 * Arg1 + Arg2 * Arg3 - 0.5

    where Arg0, Arg1, Arg2 and Arg3 are derived from

        ZERO                    the value 0
        PRIMARY_COLOR_EXT       primary color of incoming fragment
        TEXTURE                 texture color of corresponding texture unit
        CONSTANT_EXT            texture environment constant color
        PREVIOUS_EXT            result of previous texture environment; on
                                texture unit 0, this maps to PRIMARY_COLOR_EXT
        TEXTURE<n>_ARB          texture color of the <n>th texture unit

    In addition, the result may be scaled by 1.0, 2.0 or 4.0.
```

Issues

    None

New Procedures and Functions

    None

New Tokens

    Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv, and
    TexEnviv when the <pname> parameter value is TEXTURE_ENV_MODE

        COMBINE4_NV   0x8503

    Accepted by the <pname> parameter of GetTexEnvfv, GetTexEnviv,
    TexEnvf, TexEnvi, TexEnvfv, and TexEnviv when the <target> parameter
    value is TEXTURE_ENV

        SOURCE3_RGB_NV   0x8583
        SOURCE3_ALPHA_NV   0x858B
        OPERAND3_RGB_NV   0x8593
        OPERAND3_ALPHA_NV   0x859B

    Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv, and
    TexEnviv when the <pname> parameter value is SOURCE0_RGB_EXT,
    SOURCE1_RGB_EXT, SOURCE2_RGB_EXT, SOURCE3_RGB_NV, SOURCE0_ALPHA_EXT,
    SOURCE1_ALPHA_EXT, SOURCE2_ALPHA_EXT, or SOURCE3_ALPHA_NV

        ZERO
        TEXTURE<n>_ARB

    where <n> is in the range 0 to NUMBER_OF_TEXTURE_UNITS_ARB-1.

    Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv, and
    TexEnviv when the <pname> parameter value is OPERAND0_RGB_EXT,
    OPERAND1_RGB_EXT, OPERAND2_RGB_EXT or OPERAND3_RGB_NV

        SRC_COLOR
        ONE_MINUS_SRC_COLOR
        SRC_ALPHA
        ONE_MINUS_SRC_ALPHA

    Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv, and
    TexEnviv when the <pname> parameter value is OPERAND0_ALPHA_EXT,
    OPERAND1_ALPHA_EXT, OPERAND2_ALPHA_EXT, or OPERAND3_ALPHA_NV

        SRC_ALPHA
        ONE_MINUS_SRC_ALPHA

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

    None

Additions to Chapter 3 of the GL Specification (Rasterization)

    Added to subsection 3.8.9, before the paragraph describing the state
    requirements:

    If the value of TEXTURE_ENV_MODE is COMBINE4_NV, the form of the
    texture function depends on the values of COMBINE_RGB_EXT and

COMBINE_ALPHA_EXT, according to table 3.21. The RGB and ALPHA results of the texture function are then multiplied by the values of RGB_SCALE_EXT and ALPHA_SCALE, respectively. The results are clamped to [0,1]. If the value of COMBINE_RGB_EXT or COMBINE_ALPHA_EXT is not one of the listed values, the result is undefined.

COMBINE_RGB_EXT or COMBINE_ALPHA_EXT	Texture Function
ADD	Arg0 * Arg1 + Arg2 * Arg3
ADD_SIGNED_EXT	Arg0 * Arg1 + Arg2 * Arg3 - 0.5

Table 3.21: COMBINE4_NV texture functions

The arguments Arg0, Arg1, Arg2 and Arg3 are determined by the values of SOURCE<n>_RGB_EXT, SOURCE<n>_ALPHA_EXT, OPERAND<n>_RGB_EXT and OPERAND<n>_ALPHA_EXT. In the following two tables, Ct and At are the filtered texture RGB and alpha values; Cc and Ac are the texture environment RGB and alpha values; Cf and Af are the RGB and alpha of the primary color of the incoming fragment; and Cp and Ap are the RGB and alpha values resulting from the previous texture environment. On texture environment 0, Cp and Ap are identical to Cf and Af, respectively. Ct<n> and At<n> are the filtered texture RGB and alpha values from the texture bound to the <n>th texture unit. If the <n>th texture unit is disabled, the value of each component is 1. The relationship is described in tables 3.22 and 3.23.

SOURCE<n>_RGB_EXT	OPERAND<n>_RGB_EXT	Argument
ZERO	SRC_COLOR	0
	ONE_MINUS_SRC_COLOR	1
	SRC_ALPHA	0
	ONE_MINUS_SRC_ALPHA	1
TEXTURE	SRC_COLOR	Ct
	ONE_MINUS_SRC_COLOR	(1-Ct)
	SRC_ALPHA	At
	ONE_MINUS_SRC_ALPHA	(1-At)
CONSTANT_EXT	SRC_COLOR	Cc
	ONE_MINUS_SRC_COLOR	(1-Cc)
	SRC_ALPHA	Ac
	ONE_MINUS_SRC_ALPHA	(1-Ac)
PRIMARY_COLOR_EXT	SRC_COLOR	Cf
	ONE_MINUS_SRC_COLOR	(1-Cf)
	SRC_ALPHA	Af
	ONE_MINUS_SRC_ALPHA	(1-Af)
PREVIOUS_EXT	SRC_COLOR	Cp
	ONE_MINUS_SRC_COLOR	(1-Cp)
	SRC_ALPHA	Ap
	ONE_MINUS_SRC_ALPHA	(1-Ap)
TEXTURE<n>_ARB	SRC_COLOR	Ct<n>
	ONE_MINUS_SRC_COLOR	(1-Ct<n>)
	SRC_ALPHA	At<n>
	ONE_MINUS_SRC_ALPHA	(1-At<n>)

Table 3.22: Arguments for COMBINE_RGB_EXT functions

SOURCE<n>_ALPHA_EXT	OPERAND<n>_ALPHA_EXT	Argument
ZERO	SRC_ALPHA	0
	ONE_MINUS_SRC_ALPHA	1
TEXTURE	SRC_ALPHA	At
	ONE_MINUS_SRC_ALPHA	(1-At)
CONSTANT_EXT	SRC_ALPHA	Ac
	ONE_MINUS_SRC_ALPHA	(1-Ac)
PRIMARY_COLOR_EXT	SRC_ALPHA	Af
	ONE_MINUS_SRC_ALPHA	(1-Af)
PREVIOUS_EXT	SRC_ALPHA	Ap
	ONE_MINUS_SRC_ALPHA	(1-Ap)
TEXTURE<n>_ARB	SRC_ALPHA	At<n>
	ONE_MINUS_SRC_ALPHA	(1-At<n>)

Table 3.23: Arguments for COMBINE_ALPHA_EXT functions

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the GL Specification (Special Functions)

None

Additions to Chapter 6 of the GL Specification (State and State Requests)

None

Additions to the GLX Specification

None

GLX Protocol

None

Errors

INVALID_ENUM is generated if <params> value for SOURCE0_RGB_EXT, SOURCE1_RGB_EXT, SOURCE2_RGB_EXT, SOURCE3_RGB_NV, SOURCE0_ALPHA_EXT, SOURCE1_ALPHA_EXT, SOURCE2_ALPHA_EXT or SOURCE3_ALPHA_NV is not one of ZERO, TEXTURE, CONSTANT_EXT, PRIMARY_COLOR_EXT, PREVIOUS_EXT or TEXTURE<n>_ARB, where <n> is in the range 0 to NUMBER_OF_TEXTURE_UNITS_ARB-1.

INVALID_ENUM is generated if <params> value for OPERAND0_RGB_EXT, OPERAND1_RGB_EXT, OPERAND2_RGB_EXT or OPERAND3_RGB_NV is not one of SRC_COLOR, ONE_MINUS_SRC_COLOR, SRC_ALPHA or ONE_MINUS_SRC_ALPHA.

INVALID_ENUM is generated if <params> value for OPERAND0_ALPHA_EXT OPERAND1_ALPHA_EXT, OPERAND2_ALPHA_EXT, or OPERAND3_ALPHA_NV is not one of SRC_ALPHA or ONE_MINUS_SRC_ALPHA.

Modifications to EXT_texture_env_combine

This extension relaxes the restrictions on SOURCE<n>_RGB_EXT, SOURCE<n>_ALPHA_EXT, OPERAND<n>_RGB_EXT and OPERAND<n>_ALPHA_EXT for use with EXT_texture_env_combine. All params specified by Table 3.22 and Table 3.23 are valid.

Dependencies on ARB_multitexture

If ARB_multitexture is not implemented, all references to TEXTURE<n>_ARB and NUMBER_OF_TEXTURE_UNITS_ARB are deleted.

New State

Get Value	Get Command	Type	Initial Value	Attribute
SOURCE3_RGB_NV	GetTexEnviv	n x Z5+n	ZERO	texture
SOURCE3_ALPHA_NV	GetTexEnviv	n x Z5+n	ZERO	texture
OPERAND3_RGB_NV	GetTexEnviv	n x Z2	ONE_MINUS_SRC_COLOR	texture
OPERAND3_ALPHA_NV	GetTexEnviv	n x Z2	ONE_MINUS_SRC_ALPHA	texture

New Implementation Dependent State

None

NVIDIA Implementation Details

Because of a hardware limitation, TNT, TNT2, GeForce, and Quadro treat "scale by 4.0" with the COMBINE_RGB_EXT or COMBINE_ALPHA_EXT mode of ADD_SIGNED_EXT as "scale by 2.0".

Name

    NV_texture_expand_normal

Name Strings

    GL_NV_texture_expand_normal

Notice

    Copyright NVIDIA Corporation, 2002.

IP Status

    NVIDIA Proprietary.

Status

    Implemented, November 2002

Version

```
    Last Modified:      $Date: 2005/06/16 $
    NVIDIA Revision:    4
```

Number

    286

Support

    NVIDIA plans to discontinue this extension for future GPU
    architectures. Support for NV3x (GeForce FX), NV4x (GeForce 6
    Series), and G7x (GeForce 7x00) architectures will continue.

    As an alternative to the EXT_texture_expand_normal functionality,
    developers can either use the signed fixed-point texture formats
    provided by NV_texture_shader (such as GL_SIGNED_RGBA8_NV) or perform
    the "expand normal" operation with shader instructions (typically
    just a MAD)..

Dependencies

    OpenGL 1.1 is required.

Overview

    This extension provides a remapping mode where unsigned texture
    components (in the range [0,1]) can be treated as though they
    contained signed data (in the range [-1,+1]). This allows
    applications to easily encode signed data into unsigned texture
    formats.

    The functionality of this extension is nearly identical to the
    EXPAND_NORMAL_NV remapping mode provided in the NV_register_combiners
    extension, although it applies even if register combiners are used.

Issues

(1) When is the remapping applied?

RESOLVED: It would be possible to remap after loading each texel, remap after all filtering is done, or something in between. Ignoring implementation-dependent rounding errors, it really doesn't matter.

The spec language says that the remapping is applied after filtering texel values within each level. For LINEAR_MIPMAP_LINEAR, this means that the remapping is "done" twice. This approach was chosen solely to simplify the spec language, and does not necessarily reflect NVIDIA's implementation.

(2) Should the remapping mode apply to textures with signed components?

RESOLVED: No -- the EXPAND_NORMAL_NV mapping is ignored for such textures.

(3) NV_texture_shader provides several internal formats with a mix of signed and unsigned components. For example, the base formats DSDT_MAG_NV, and DSDT_MAG_INTENSITY_NV have this property, and there is a variant of RGBA where the RGB components are signed, but the A component is unsigned. What should happen in this case?

RESOLVED: The unsigned components are remapped; the signed components are unmodified.

(4) What should be said about signed fixed-point precision and range of actual implementations?

RESOLVED: The fundamental problem is that it is not possible to derive a linear mapping taking unsigned values that exactly represents -1.0, 0.0, and +1.0.

The mapping chosen for current NVIDIA implementations does not exactly represent +1.0. For an n-bit fixed-point component, 0 maps to -1.0, 2^(n-1) maps to 0.0, and 2^n-1 (maximum value) maps to 1.0 - 1/(2^(n-1)). This same conversion is applied to stored textures using the signed texture types in NV_texture_shader.

This specification is written using the conventional OpenGL mapping where -1.0 and +1.0 can be represented exactly, but 0.0 can not. The specification is simpler and avoids precision-dependent language describing the mapping. We expect some leeway in how the remapping is applied.

This issue is discussed in more detail in the issues section of the NV_texture_shader specification (the question is phrased identically).

(5) Are texture border color components remapped?

RESOLVED: Yes -- if the border values are used for filtering, border color components are remapped identically to normal texel components.

New Procedures and Functions

None.

New Tokens

Accepted by the <pname> parameters of TexParameteri, TexParameteriv, TexParameterf, TexParameterfv, GetTexParameteri, and GetTexParameteriv:

```
TEXTURE_UNSIGNED_REMAP_MODE_NV               0x888F
```

Additions to Chapter 2 of the OpenGL 1.4 Specification (OpenGL Operation)

None.

Additions to Chapter 3 of the OpenGL 1.4 Specification (Rasterization)

Modify Section 3.8.4, Texture Parameters, p.135

(modify Table 3.19, p. 137)

Name	Type	Legal Values
TEXTURE_UNSIGNED_ REMAP_MODE_NV	enum	EXPAND_NORMAL_NV, NONE

Modify Section 3.8.8, Texture Minification, p.140

(add after the last paragraph before the "Mipmapping" subsection, p. 144)

After the texture filter is applied, the filtered texture values are optionally rescaled, converting unsigned texture components encoded in the range [0,1] to signed values in the range [-1,+1]. If the texture parameter TEXTURE_UNSIGNED_REMAP_MODE_NV is EXPAND_NORMAL_NV, the filtered values for each unsigned component of the texture is transformed by

```
tau = 2 * tau - 1.
```

For components

Additions to Chapter 4 of the OpenGL 1.4 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 1.4 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 1.4 Specification (State and State Requests)

None.

Additions to Appendix A of the OpenGL 1.4 Specification (Invariance)

None.

Additions to the AGL/GLX/WGL Specifications

None.

GLX Protocol

None.

Errors

None.

New State

(add to table 6.15, p. 230)

Get Value	Type	Get Command	Initial Value	Description	Sec.	Attribute
TEXTURE_UNSIGNED_REMAP_MODE_NV	nxZ2	GetTexParameteriv	NONE	unsigned component remapping	3.8.8	texture

Name

    NV_texture_rectangle

Name Strings

    GL_NV_texture_rectangle

Notice

Status

    Implemented in NVIDIA's Release 10 drivers.

Version

    NVIDIA Date: March 5, 2004
    $Id: //sw/main/docs/OpenGL/specs/GL_NV_texture_rectangle.txt#6 $

Number

    229

Dependencies

    Written based on the OpenGL 1.2.1 specification including
    ARB_texture_cube_map wording.

    IBM_mirrored_repeat affects the definition of this extension.

    ARB_texture_border_clamp affects the definition of this extension.

    EXT_paletted_texture affects the definition of this extension.

    This extension affects the definition of the NV_texture_shader
    extension.

Overview

    OpenGL texturing is limited to images with power-of-two dimensions
    and an optional 1-texel border. NV_texture_rectangle extension
    adds a new texture target that supports 2D textures without requiring
    power-of-two dimensions.

    Non-power-of-two dimensioned textures are useful for storing
    video images that do not have power-of-two dimensions. Re-sampling
    artifacts are avoided and less texture memory may be required by using
    non-power-of-two dimensioned textures. Non-power-of-two dimensioned
    textures are also useful for shadow maps and window-space texturing.

    However, non-power-of-two dimensioned (NPOTD) textures have
    limitations that do not apply to power-of-two dimensioned (POT)
    textures. NPOTD textures may not use mipmap filtering; POTD
    textures support both mipmapped and non-mipmapped filtering.
    NPOTD textures support only the GL_CLAMP, GL_CLAMP_TO_EDGE,

and GL_CLAMP_TO_BORDER_ARB wrap modes; POTD textures support GL_CLAMP_TO_EDGE, GL_REPEAT, GL_CLAMP, GL_MIRRORED_REPEAT_IBM, and GL_CLAMP_TO_BORDER. NPOTD textures do not support an optional 1-texel border; POTD textures do support an optional 1-texel border.

NPOTD textures are accessed by non-normalized texture coordinates. So instead of thinking of the texture image lying in a [0..1]x[0..1] range, the NPOTD texture image lies in a [0..w]x[0..h] range.

This extension adds a new texture target and related state (proxy, binding, max texture size).

Issues

Should rectangular textures simply be an extension to the 2D texture target that allows non-power-of-two widths and heights?

RESOLUTION: No. The rectangular texture is an entirely new texture target type called GL_TEXTURE_RECTANGLE_NV. This is because while the texture rectangle target relaxes the power-of-two dimensions requirements of the texture 2D target, it also has limitations such as the absence of both mipmapping and the GL_REPEAT and GL_MIRRORED_REPEAT_IBM wrap modes. Additionally, rectangular textures do not use [0..1] normalized texture coordinates.

How is the image of a rectangular texture specified?

RESOLUTION: Using the standard OpenGL API for specifying a 2D texture image: glTexImage2D, glSubTexImage2D, glCopyTexImage2D, and glCopySubTexImage2D. The target for these commands is GL_TEXTURE_RECTANGLE_NV though.

This is similar to how the ARB_texture_cube_map extension uses the 2D texture image specification API though with its own texture target.

Should 3D textures be allowed to be NPOTD?

RESOLUTION: No. That should be left to another extension.

Should cube map textures be allowed to be NPOTD?

RESOLUTION: No. Probably not particularly interesting for cube maps. If it becomes important, another extension should provide NPOTD cube maps.

Should 1D textures be allowed to be NPOTD?

RESOLUTION: No. Rectangular textures are always considered 2D by this extension. You can always simulate a 1D NPOTD textures by using a 2D Wx1 or 1xH dimensioned rectangular texture.

Should anything be said about performance?

RESOLUTION: No, but developers should not be surprised if conventional POTD textures will render slightly faster than NPOTD

textures. This is particularly likely to be true when NPOTD
textures are minified leading to texture cache thrashing.

How are rectangular textures enabled?

RESOLUTION: Since rectangular textures add a new texture target,
you enable rectangular textures by enabling this target. Example:

```
glEnable(GL_TEXTURE_RECTANGLE_NV);
```

What is the priority of the rectangular texture target enable relative to existing texture enables?

RESOLUTION: The texture rectangle target is like a 2D texture in
many ways so its enable priority is just above GL_TEXTURE_2D. From
lowest priority to highest priority: GL_TEXTURE_1D, GL_TEXTURE_2D,
GL_TEXTURE_RECTANGLE_NV, GL_TEXTURE_3D, GL_TEXTURE_CUBE_MAP_ARB.

What is the default wrap state for a texture rectangle?

RESOLUTION: GL_CLAMP_TO_EDGE. The normal default wrap state is
GL_REPEAT, but that mode is not allowed for rectangular textures?

What is the default minification filter for a texture rectangle?

RESOLUTION: GL_LINEAR. The normal default minification filter
state is GL_NEAREST_MIPMAP_LINEAR, but that mode is not allowed
for rectangular textures because mipmapping is not supported.

Do paletted textures work with rectangular textures?

RESOLUTION: No. Similar (but not identical) functionality can
be accomplished using dependent texture shader operations (see
NV_texture_shader).

The difference between paletted texture accesses and dependent
texture accesses is that paletted texture lookups are
"pre-filtering" while dependent texture shader operations are
"post-filtering".

Can compressed texture images be specified for a rectangular texture?

RESOLUTION: The generic texture compression internal formats
introduced by ARB_texture_compression are supported for rectangular
textures because the image is not presented as compressed data and
the ARB_texture_compression extension always permits generic texture
compression internal formats to be stored in uncompressed form.
Implementations are free to support generic compression internal
formats for rectangular textures if supported but such support is
not required.

This extensions makes a blanket statement that specific compressed
internal formats for use with CompressedTexImage<n>DARB are NOT
supported for rectangular textures. This is because several
existing hardware implementations of texture compression formats
such as S3TC are not designed for compressing rectangular textures.
This does not preclude future texture compression extensions from

supporting compressed internal formats that do work with rectangular extensions (by relaxing the current blanket error condition).

Does this extension work with SGIX_shadow-style shadow mapping?

RESOLUTION: Yes. The one non-obvious allowance to support SGIX_shadow-style shadow mapping is that the R texture coordinate wrap mode remains UNCHANGED for rectangular textures. Clamping of the R texture coordinate for rectangular textures uses the standard [0,1] interval rather than the [0,ws] or [0,hs] intervals as in the case of S and T. This is because R represents a depth value in the [0,1] range whether using a 2D or rectangular texture.

New Procedures and Functions

None

New Tokens

Accepted by the <cap> parameter of Enable, Disable, IsEnabled, and by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev, and by the <target> parameter of BindTexture, GetTexParameterfv, GetTexParameteriv, TexParameterf, TexParameteri, TexParameterfv, and TexParameteriv:

```
    TEXTURE_RECTANGLE_NV               0x84F5
```

Accepted by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

```
    TEXTURE_BINDING_RECTANGLE_NV       0x84F6
```

Accepted by the <target> parameter of GetTexImage, GetTexLevelParameteriv, GetTexLevelParameterfv, TexImage2D, CopyTexImage2D, TexSubImage2D, and CopySubTexImage2D:

```
    TEXTURE_RECTANGLE_NV
```

Accepted by the <target> parameter of GetTexLevelParameteriv, GetTexLevelParameterfv, GetTexParameteriv, and TexImage2D:

```
    PROXY_TEXTURE_RECTANGLE_NV          0x84F7
```

Accepted by the <pname> parameter of GetBooleanv, GetDoublev, GetIntegerv, and GetFloatv:

```
    MAX_RECTANGLE_TEXTURE_SIZE_NV       0x84F8
```

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

None

Additions to Chapter 3 of the GL Specification (Rasterization)

-- Section 3.6.3 "Pixel Transfer Modes" under "Color Table Specification" or the ColorTableEXT description in the EXT_paletted_texture specification (rev 1.2)

Add the following statement after introducing ColorTableEXT:

"The error INVALID_ENUM is generated if the target to ColorTable (or ColorTableEXT or the various ColorTable and ColorTableEXT alternative commands) is TEXTURE_RECTANGLE_NV or PROXY_TEXTURE_RECTANGLE_NV."

-- Section 3.8.1 "Texture Image Specification"

Change the second sentence through the rest of the paragraph describing TexImage2D on page 116 to:

"<target> must be one of TEXTURE_2D for a 2D texture, or one of TEXTURE_RECTANGLE_NV for a rectangle texture, or one of TEXTURE_CUBE_MAP_POSITIVE_X_ARB, TEXTURE_CUBE_MAP_NEGATIVE_X_ARB, TEXTURE_CUBE_MAP_POSITIVE_Y_ARB, TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB, TEXTURE_CUBE_MAP_POSITIVE_Z_ARB, or TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB for a cube map texture. Additionally, <target> can be either PROXY_TEXTURE_2D for a 2D proxy texture or PROXY_TEXTURE_RECTANGLE_NV for a rectangle proxy texture or PROXY_TEXTURE_CUBE_MAP_ARB for a cube map proxy texture as discussed in section 3.8.7. The other parameters match the corresponding parameters of TexImage3D."

Add a following paragraph reading:

"Rectangular textures do not support paletted formats. The error INVALID_ENUM is generated if the target is TEXTURE_RECTANGLE_NV or PROXY_TEXTURE_RECTANGLE_NV and the format is COLOR_INDEX or the internalformat is COLOR_INDEX or one of the COLOR_INDEX<n>_EXT internal formats."

Change the 14th paragraph (page 116) to read:

"In a similar fashion, the maximum allowable width of a rectangular texture image, and the maximum allowable height of a rectangular texture image, must be at least the implementation-dependent value of MAX_RECTANGLE_TEXTURE_SIZE_NV."

Add the following paragraph after the paragraph introducing TexImage2D (page 116):

"When the target is TEXTURE_RECTANGLE_NV, the INVALID_VALUE error is generated if border is any value other than zero or the level is any value other than zero. Also when the target is TEXTURE_RECTANGLE_NV, the texture dimension restrictions specified by equations 3.11, 3.12, and 3.13 are ignored; however, if the width is less than zero or the height is less than zero, the error INVALID_VALUE is generated. In the case of a rectangular texture, ws and hs equal the specified width and height respectively of the rectangular texture image while ds is 1."

Amend the following paragraph that was added by the

ARB_texture_cube_map specification after the first paragraph on page 117:

"A 2D texture consists of a single 2D texture image. A rectangle texture consists of a single 2D texture image. A cube map texture is a set of six 2D texture images. The six cube map texture targets form a single cube map texture though each target names a distinct face of the cube map. The TEXTURE_CUBE_MAP_*_ARB targets listed above update their appropriate cube map face 2D texture image. Note that the six cube map 2D image tokens such as TEXTURE_CUBE_MAP_POSITIVE_X_ARB are used when specifying, updating, or querying one of a cube map's six 2D image, but when enabling cube map texturing or binding to a cube map texture object (that is when the cube map is accessed as a whole as opposed to a particular 2D image), the TEXTURE_CUBE_MAP_ARB target is specified."

Append to the end of the third to the last paragraph in the section (page 118):

"A rectangular texture array has depth dt=1, with height ht and width wt defined by the specified image height and width parameters."

-- Section 3.8.2 "Alternate Texture Image Specification Commands"

Add TEXTURE_RECTANGLE_NV to the second paragraph (page 120) to say:

... "Currently, <target> must be TEXTURE_2D, TEXTURE_RECTANGLE_NV, TEXTURE_CUBE_MAP_POSITIVE_X_ARB, TEXTURE_CUBE_MAP_NEGATIVE_X_ARB, TEXTURE_CUBE_MAP_POSITIVE_Y_ARB, TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB, TEXTURE_CUBE_MAP_POSITIVE_Z_ARB, or TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB." ...

Add TEXTURE_RECTANGLE_NV to the fourth paragraph (page 121) to say:

... "Currently the target arguments of TexSubImage1D and CopyTexSubImage1D must be TEXTURE_1D, the <target> arguments of TexSubImage2D and CopyTexSubImage2D must be one of TEXTURE_2D, TEXTURE_RECTANGLE_NV, TEXTURE_CUBE_MAP_POSITIVE_X_ARB, TEXTURE_CUBE_MAP_NEGATIVE_X_ARB, TEXTURE_CUBE_MAP_POSITIVE_Y_ARB, TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB, TEXTURE_CUBE_MAP_POSITIVE_Z_ARB, or TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB, and the <target> arguments of TexSubImage3D and CopyTexSubImage3D must be TEXTURE_3D." ...

Also add to the end of the fourth paragraph (121):

"If target is TEXTURE_RECTANGLE_NV and level is not zero, the error INVALID_VALUE is generated."

-- Section "Compressed Texture Images" in the ARB_texture_compression specification

Add the following paragraph after introducing the CompressedTexImage<n>DARB commands:

"The error INVALID_ENUM is generated if the target parameter to one of the CompressedTexImage<n>DARB commands is TEXTURE_RECTANGLE_NV."

Add the following paragraph after introducing the CompressedTexSubImage<n>DARB commands:

"The error INVALID_ENUM is generated if the target parameter to one of the CompressedTexSubImage<n>DARB commands is TEXTURE_RECTANGLE_NV."

-- Section 3.8.3 "Texture Parameters"

Add TEXTURE_RECTANGLE_NV to paragraph one (page 124) to say:

... "<target> is the target, either TEXTURE_1D, TEXTURE_2D, TEXTURE_RECTANGLE_NV, TEXTURE_3D, or TEXTURE_CUBE_MAP_ARB." ...

Add the following paragraph to the end of the section (page 134):

"Certain texture parameter values may not be specified for textures with a target of TEXTURE_RECTANGLE_NV. The error INVALID_ENUM is generated if the target is TEXTURE_RECTANGLE_NV and the TEXTURE_WRAP_S, TEXTURE_WRAP_T, or TEXTURE_WRAP_R parameter is set to REPEAT or MIRRORED_REPEAT_IBM. The error INVALID_ENUM is generated if the target is TEXTURE_RECTANGLE_NV and the TEXTURE_MIN_FILTER is set to a value other than NEAREST or LINEAR (no mipmap filtering is permitted). The error INVALID_ENUM is generated if the target is TEXTURE_RECTANGLE_NV and TEXTURE_BASE_LEVEL is set to any value other than zero."

-- Section 3.8.4 "Texture Wrap Modes"

Add this final additional paragraph:

"Texture coordinates are clamped differently for rectangular textures. The r texture coordinate is wrapped as described above (as required for shadow mapping to operate correctly). When the texture target is TEXTURE_RECTANGLE_NV, the s and t coordinates are wrapped as follows: CLAMP causes the s coordinate to be clamped to the range [0,ws]. CLAMP causes the t coordinate to be clamped to the range [0,hs]. CLAMP_TO_EDGE causes the s coordinate to be clamped to the range [0.5,ws-0.5]. CLAMP_TO_EDGE causes the t coordinate to be clamped to the range [0.5,hs-0.5]. CLAMP_TO_BORDER_ARB causes the s coordinate to be clamped to the range [-0.5,ws+0.5]. CLAMP_TO_BORDER_ARB causes the t coordinate to be clamped to the range [-0.5,hs+0.5]."

-- Section 3.8.5 "Texture Minification" under "Mipmapping"

Change the second full paragraph on page 126 to read:

"For non-rectangular textures, let u(x,y) = 2^n*s(x,y), v(x,y) = 2^m*t(x,y), and w(x,y) = 2^l*r(x,y), where n, m, and l are defined by equations 3.11, 3.12, and 3.13 with ws, hs, and ds equal to the width, height, and depth of the image array whose level is TEXTURE_BASE_LEVEL. However, for rectangular textures let u(x,y) = s(x,y), v(x,y) = t(x,y), and w(x,y) = r(x,y)."

Update the last sentence in the first full paragraph on page 127 to read:

```
    "Depending on whether the texture's target is rectangular or
    non-rectangular, this means the texel at location (i,j,k) becomes
    the texture value, with i given by

             /  floor(u),    s < 1
            /
       i = {    2^n-1,       s == 1, non-rectangular texture    (3.17)
            \
             \  ws-1,        s == 1, rectangular texture

    (Recall that if TEXTURE_WRAP_S is REPEAT, then 0 <= s < 1.)  Similarly,
    j is found as

             /  floor(v),    t < 1
            /
       j = {    2^m-1,       t == 1, non-rectangular texture    (3.18)
            \
             \  hs-1,        t == 1, rectangular texture

    and k is found as

             /  floor(w),    r < 1
            /
       k = {    2^l-1,       r == 1, non-rectangular texture     (3.19)
            \
             \  0,           r == 1, rectangular texture"

    Change the last sentence in the partial paragraph after equation
    3.19 to read:

    "For a two-dimensional or rectangular texture, k is irrelevant;
    the texel at location (i,j) becomes the texture value."

    Change the sentence preceding equation 3.20 (page 128) specifying
    how to compute the value tau for a two-dimensional texture to:

    "For a two-dimensional or rectangular texture,"

    Follow the first full paragraph on page 130 with:

    "Rectangular textures do not support mipmapping (it is an error to
    specify a minification filter that requires mipmapping)."
```

-- Section 3.8.7 "Texture State and Proxy State"

```
    Change the first sentence of the first paragraph (page 131) to say:

    "The state necessary for texture can be divided into two categories.
    First, there are the ten sets of mipmap arrays (one each for the
    one-, two-, and three-dimensional texture targets, one for the
    rectangular texture target (though the rectangular texture target
    has only one mipmap level), and six for the cube map texture targets)
    and their number." ...

    Change the fourth and third to last sentences of the first paragraph
    to say:
```

"In the initial state, the value assigned to TEXTURE_MIN_FILTER is NEAREST_MIPMAP_LINEAR, except for rectangular textures where the initial value is LINEAR, and the value for TEXTURE_MAG_FILTER is LINEAR. s, t, and r warp modes are all set to REPEAT, except for rectangular textures where the initial value is CLAMP_TO_EDGE."

Change the second paragraph (page 132) to say:

"In addition to the one-, two-, three-dimensional, rectangular, and the six cube map sets of image arrays, the partially instantiated one-, two-, and three-dimensional, rectangular, and one cube map sets of proxy image arrays are maintained." ...

Change the third paragraph (page 132) to:

"One- and two-dimensional and rectangular proxy arrays are operated on in the same way when TexImage1D is executed with target specified as PROXY_TEXTURE_1D, or TexImage2D is executed with target specified as PROXY_TEXTURE_2D or PROXY_TEXTURE_RECTANGLE_NV."

Change the second sentence of the fourth paragraph (page 132) to:

"Therefore PROXY_TEXTURE_1D, PROXY_TEXTURE_2D, PROXY_TEXTURE_RECTANGLE_NV, PROXY_TEXTURE_3D, and PROXY_TEXTURE_CUBE_MAP_ARB cannot be used as textures, and their images must never be queried using GetTexImage." ...

-- Section 3.8.8 "Texture Objects"

Change the first sentence of the first paragraph (page 132) to say:

"In addition to the default textures TEXTURE_1D, TEXTURE_2D, TEXTURE_RECTANGLE_NV, TEXTURE_3D, and TEXTURE_CUBE_MAP_ARB, named one-dimensional, two-dimensional, rectangular, and three-dimensional texture objects and cube map texture objects can be created and operated on." ...

Change the second paragraph (page 132) to say:

"A texture object is created by binding an unused name to TEXTURE_1D, TEXTURE_2D, TEXTURE_RECTANGLE_NV, TEXTURE_3D, or TEXTURE_CUBE_MAP_ARB." ... "If the new texture object is bound to TEXTURE_1D, TEXTURE_2D, TEXTURE_RECTANGLE_NV, TEXTURE_3D, or TEXTURE_CUBE_MAP_ARB, it remains a one-dimensional, two-dimensional, rectangular, three-dimensional, or cube map texture until it is deleted."

Change the third paragraph (page 133) to say:

"BindTexture may also be used to bind an existing texture object to either TEXTURE_1D, TEXTURE_2D, TEXTURE_RECTANGLE_NV, TEXTURE_3D, or TEXTURE_CUBE_MAP_ARB."

Change paragraph five (page 133) to say:

"In the initial state, TEXTURE_1D, TEXTURE_2D, TEXTURE_RECTANGLE_NV, TEXTURE_3D, and TEXTURE_CUBE_MAP have one-dimensional, two-dimensional, rectangular, three-dimensional, and cube map state vectors associated with them respectively." ... "The initial, one-dimensional, two-dimensional, rectangular, three-dimensional, and cube map texture is therefore operated upon, queried, and applied as TEXTURE_1D, TEXTURE_2D, TEXTURE_RECTANGLE_NV, TEXTURE_3D, and TEXTURE_CUBE_MAP_ARB respectively while 0 is bound to the corresponding targets."

Change paragraph six (page 133) to say:

... "If a texture that is currently bound to one of the targets TEXTURE_1D, TEXTURE_2D, TEXTURE_RECTANGLE_NV, TEXTURE_3D, or TEXTURE_CUBE_MAP_ARB is deleted, it is as though BindTexture has been executed with the same <target> and <texture> zero." ...

-- Section 3.8.10 "Texture Application"

Replace the beginning sentences of the first paragraph (page 138) with:

"Texturing is enabled or disabled using the generic Enable and Disable commands, respectively, with the symbolic constants TEXTURE_1D, TEXTURE_2D, TEXTURE_RECTANGLE_NV, TEXTURE_3D, or TEXTURE_CUBE_MAP_ARB to enable the one-dimensional, two-dimensional, rectangular, three-dimensional, or cube map texturing respectively. If both two- and one-dimensional textures are enabled, the two-dimensional texture is used. If the rectangular and either of the two- or one-dimensional textures is enabled, the rectangular texture is used. If the three-dimensional and any of the rectangular, two-dimensional, or one-dimensional textures is enabled, the three-dimensional texture is used. If the cube map texture and any of the three-dimensional, rectangular, two-dimensional, or one-dimensional textures is enabled, then cube map texturing is used.

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the GL Specification (Special Functions)

-- Section 5.4 "Display Lists"

In the first paragraph (page 179), add PROXY_TEXTURE_RECTANGLE_NV to the list of PROXY_* tokens.

Additions to Chapter 6 of the GL Specification (State and State Requests)

-- Section 6.1.3 "Enumerated Queries"

Change the fourth paragraph (page 183) to say:

"The GetTexParameter parameter <target> may be one of TEXTURE_1D, TEXTURE_2D, TEXTURE_RECTANGLE_NV, TEXTURE_3D, or TEXTURE_CUBE_MAP_ARB, indicating the currently bound one-dimensional,

two-dimensional, rectangular, three-dimensional, or cube map
texture object. For GetTexLevelParameter, <target> may be one
of TEXTURE_1D, TEXTURE_2D, TEXTURE_RECTANGLE_NV, TEXTURE_3D,
TEXTURE_CUBE_MAP_POSITIVE_X_ARB, TEXTURE_CUBE_MAP_NEGATIVE_X_ARB,
TEXTURE_CUBE_MAP_POSITIVE_Y_ARB, TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB,
TEXTURE_CUBE_MAP_POSITIVE_Z_ARB, TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB,
PROXY_TEXTURE_1D, PROXY_TEXTURE_2D, PROXY_TEXTURE_RECTANGLE_NV,
PROXY_TEXTURE_3D, or PROXY_TEXTURE_CUBE_MAP_ARB, indicating the
one-dimensional texture object, two-dimensional texture object,
rectangular texture object, three-dimensional texture object, or one
of the six distinct 2D images making up the cube map texture object
or one-dimensional, two-dimensional, rectangular, three-dimensional,
or cube map proxy state vector. Note that TEXTURE_CUBE_MAP_ARB is
not a valid <target> parameter for GetTexLevelParameter because it
does not specify a particular cube map face."

-- Section 6.1.4 "Texture Queries"

Change the first paragraph (page 184) to read:

... "It is somewhat different from the other get commands; <tex> is a
symbolic value indicating which texture (or texture face in the case
of a cube map texture target name) is to be obtained. TEXTURE_1D
indicates a one-dimensional texture, TEXTURE_2D indicates a
two-dimensional texture, TEXTURE_RECTANGLE_NV indicates a rectangular
texture, TEXTURE_3D indicates a three-dimensional texture, and
TEXTURE_CUBE_MAP_POSITIVE_X_ARB, TEXTURE_CUBE_MAP_NEGATIVE_X_ARB,
TEXTURE_CUBE_MAP_POSITIVE_Y_ARB, TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB,
TEXTURE_CUBE_MAP_POSITIVE_Z_ARB, and TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB
indicate the respective face of a cube map texture."

Add a final sentence to the fourth paragraph:

"Calling GetTexImage with a lod not zero when the tex is
TEXTURE_RECTANGLE_NV causes the error INVALID_VALUE."

Additions to the GLX Specification

None

GLX Protocol

None

Dependencies on ARB_texture_border_clamp

If ARB_texture_border_clamp is not supported, references to the
CLAMP_TO_BORDER_ARB wrap mode in this document should be ignored.

Dependencies on IBM_mirrored_repeat

If IBM_mirrored_repeat is not supported, references to the
MIRRORED_REPEAT_IBM wrap mode in this document should be ignored.

Dependencies on EXT_paletted_texture

If EXT_paletted_texture is not supported, references to the

COLOR_INDEX, COLOR_INDEX<n>_EXT, ColorTable, and ColorTableEXT should be ignored.

Dependencies on EXT_texture_compression_s3tc

If EXT_texture_compression_s3tc is not supported, references to CompressedTexImage2DARB and CompressedTexSubImageARB and the COMPRESSED_*_S3TC_DXT*_EXT enumerants should be ignored.

Errors

INVALID_ENUM is generated when ColorTable (or ColorTableEXT or the various ColorTable and ColorTableEXT alternative commands) is called and the target is TEXTURE_RECTANGLE_NV or PROXY_TEXTURE_RECTANGLE_NV.

INVALID_ENUM is generated when TexImage2D is called and the target is TEXTURE_RECTANGLE_NV or PROXY_TEXTURE_RECTANGLE_NV and the format is COLOR_INDEX or the internalformat is COLOR_INDEX or one of the COLOR_INDEX<n>_EXT internal formats.

INVALID_VALUE is generated when TexImage2D is called when the target is TEXTURE_RECTANGLE_NV if border is any value other than zero or the level is any value other than zero.

INVALID_VALUE is generated when TexImage2D is called when the target is TEXTURE_RECTANGLE_NV if the width is less than zero or the height is less than zero.

INVALID_VALUE is generated when TexSubImage2D or CopyTexSubImage2D is called when the target is TEXTURE_RECTANGLE_NV if the level is any value other than zero.

INVALID_ENUM is generated when one of the CompressedTexImage<n>DARB commands is called when the target parameter is TEXTURE_RECTANGLE_NV.

INVALID_ENUM is generated when one of the CompressedTexSubImage<n>DARB commands is called when the target parameter is TEXTURE_RECTANGLE_NV.

INVALID_ENUM is generated when TexParameter is called with a target of TEXTURE_RECTANGLE_NV and the TEXTURE_WRAP_S, TEXTURE_WRAP_T, or TEXTURE_WRAP_R parameter is set to REPEAT or MIRRORED_REPEAT_IBM.

INVALID_ENUM is generated when TexParameter is called with a target of TEXTURE_RECTANGLE_NV and the TEXTURE_MIN_FILTER is set to a value other than NEAREST or LINEAR.

INVALID_VALUE is generated when TexParameter is called with a target of TEXTURE_RECTANGLE_NV and the TEXTURE_BASE_LEVEL is set to any value other than zero.

INVALID_VALUE is generated when GetTexImage is called with a lod not zero when the tex is TEXTURE_RECTANGLE_NV.

New State

(table 6.12, p202) amend/add the following entries:

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
TEXTURE_RECTANGULAR_NV	B	IsEnabled	False	True if rectangular texturing is enabled	3.8.10	texture/enable
TEXTURE_BINDING_RECTANGLE_NV	Z+	GetIntegerv	0	Texture object for texture rectangle	3.8.8	texture
TEXTURE_RECTANGLE_NV	I	GetTexImage	see 3.8	rectangular texture image for lod 0	3.8	-

(table 6.13, p203) amend/add the following entries:

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
TEXTURE_MIN_FILTER	2+xZ6	GetTexparameter	NEAREST_MIPMAP_LINEAR except for rectangular which is LINEAR	Texture minification function	3.8.5	texture
TEXTURE_WRAP_S	5+xZ5	GetTexParameter	REPEAT except for rectangular which is CLAMP_TO_EDGE	Texture wrap mode S	3.8	texture
TEXTURE_WRAP_T	5+xZ5	GetTexParameter	REPEAT except for rectangular which is CLAMP_TO_EDGE	Texture wrap mode T	3.8	texture
TEXTURE_WRAP_R	5+xZ5	GetTexParameter	REPEAT except for rectangular which is CLAMP_TO_EDGE	Texture wrap mode R	3.8	texture

New Implementation Dependent State

(table 6.24, p214) add the following entry:

Get Value	Type	Get Command	Minimum Value	Description	Sec	Attribute
MAX_RECTANGLE_TEXTURE_SIZE_NV	Z+	GetIntegerv	64	Maximum rectangular texture image dimension	3.8.1	-

Revision History

Jan 2, 2003 - Fix typo in 4th paragraph of Overview to read: "NPOTD textures are accessed by non-normalized texture coordinates."

March 5, 2004 - Delete update to the convolution section because it was bogus language in the OpenGL 1.2.1 specification saying convolution affects glGetTexImage (it does not); this language was deleted in OpenGL 1.3. Fix minor typo in 6.12 table.

Name

    NV_texture_shader

Name Strings

    GL_NV_texture_shader

Notice

    Copyright NVIDIA Corporation, 1999, 2000, 2001, 2002, 2004.

IP Status

    NVIDIA Proprietary.

Status

    Shipping (since GeForce3)

Version

    NVIDIA Date: March 13, 2007
    $Id: //sw/main/docs/OpenGL/specs/GL_NV_texture_shader.txt#30 $

Number

    230

Dependencies

    Written based on the wording of the OpenGL 1.2.1 specification.

    Requires support for the ARB_multitexture extension.

    Requires support for the ARB_texture_cube_map extension.

    NV_register_combiners affects the definition of this extension.

    EXT_texture_lod_bias trivially affects the definition of this
    extension.

    ARB_texture_env_combine and/or EXT_texture_env_combine affect the
    definition of this extension.

    NV_texture_env_combine4 affects the definition of this extension.

    ARB_texture_env_add and/or EXT_texture_env_add affect the definition
    of this extension.

    NV_texture_rectangle affects the definition of this extension.

    NV_texture_shader2 depends on the definition of this extension.

    ARB_color_buffer_float affects the definiton of this extension.

Overview

Standard OpenGL and the ARB_multitexture extension define a straightforward direct mechanism for mapping sets of texture coordinates to filtered colors. This extension provides a more functional mechanism.

OpenGL's standard texturing mechanism defines a set of texture targets. Each texture target defines how the texture image is specified and accessed via a set of texture coordinates. OpenGL 1.0 defines the 1D and 2D texture targets. OpenGL 1.2 (and/or the EXT_texture3D extension) defines the 3D texture target. The ARB_texture_cube_map extension defines the cube map texture target. Each texture unit's texture coordinate set is mapped to a color using the unit's highest priority enabled texture target.

This extension introduces texture shader stages. A sequence of texture shader stages provides a more flexible mechanism for mapping sets of texture coordinates to texture unit RGBA results than standard OpenGL.

When the texture shader enable is on, the extension replaces the conventional OpenGL mechanism for mapping sets of texture coordinates to filtered colors with this extension's sequence of texture shader stages.

Each texture shader stage runs one of 21 canned texture shader programs. These programs support conventional OpenGL texture mapping but also support dependent texture accesses, dot product texture programs, and special modes. (3D texture mapping texture shader operations are NOT provided by this extension; 3D texture mapping texture shader operations are added by the NV_texture_shader2 extension that is layered on this extension. See the NV_texture_shader2 specification.)

To facilitate the new texture shader programs, this extension introduces several new texture formats and variations on existing formats. Existing color texture formats are extended by introducing new signed variants. Two new types of texture formats (beyond colors) are also introduced. Texture offset groups encode two signed offsets, and optionally a magnitude or a magnitude and an intensity. The new HILO (pronounced high-low) formats provide possibly signed, high precision (16-bit) two-component textures.

Each program takes as input the stage's interpolated texture coordinate set (s,t,r,q). Each program generates two results: a shader stage result that may be used as an input to subsequent shader stage programs, and a texture unit RGBA result that becomes the texture color used by the texture unit's texture environment function or becomes the initial value for the corresponding texture register for register combiners. The texture unit RGBA result is always an RGBA color, but the shader stage result may be one of an RGBA color, a HILO value, a texture offset group, a floating-point value, or an invalid result. When both results are RGBA colors, the shader stage result and the texture unit RGBA result are usually identical (though not in all cases).

```
    Additionally, certain programs have a side-effect such as culling
    the fragment or replacing the fragment's depth value.

    The twenty-one programs are briefly described:

    <none>

    1.   NONE - Always generates a (0,0,0,0) texture unit RGBA result.
         Equivalent to disabling all texture targets in conventional
         OpenGL.

    <conventional textures>

    2.   TEXTURE_1D - Accesses a 1D texture via (s/q).

    3.   TEXTURE_2D - Accesses a 2D texture via (s/q,t/q).

    4.   TEXTURE_RECTANGLE_NV - Accesses a rectangular texture via (s/q,t/q).

    5.   TEXTURE_CUBE_MAP_ARB - Accesses a cube map texture via (s,t,r).

    <special modes>

    6.   PASS_THROUGH_NV - Converts a texture coordinate (s,t,r,q)
         directly to a [0,1] clamped (r,g,b,a) texture unit RGBA result.

    7.   CULL_FRAGMENT_NV - Culls the fragment based on the whether each
         (s,t,r,q) is "greater than or equal to zero" or "less than zero".

    <offset textures>

    8.   OFFSET_TEXTURE_2D_NV - Transforms the signed (ds,dt) components
         of a previous texture unit by a 2x2 floating-point matrix and
         then uses the result to offset the stage's texture coordinates
         for a 2D non-projective texture.

    9.   OFFSET_TEXTURE_2D_SCALE_NV - Same as above except the magnitude
         component of the previous texture unit result scales the red,
         green, and blue components of the unsigned RGBA texture 2D
         access.

    10.  OFFSET_TEXTURE_RECTANGLE_NV - Similar to OFFSET_TEXTURE_2D_NV
         except that the texture access is into a rectangular
         non-projective texture.

    11.  OFFSET_TEXTURE_RECTANGLE_SCALE_NV - Similar to
         OFFSET_TEXTURE_2D_SCALE_NV except that the texture access is
         into a rectangular non-projective texture.
    <dependent textures>

    12.  DEPENDENT_AR_TEXTURE_2D_NV - Converts the alpha and red
         components of a previous shader result into an (s,t) texture
         coordinate set to access a 2D non-projective texture.

    13.  DEPENDENT_GB_TEXTURE_2D_NV - Converts the green and blue
         components of a previous shader result into an (s,t) texture
         coordinate set to access a 2D non-projective texture.

    <dot product textures>

    14.  DOT_PRODUCT_NV - Computes the dot product of the texture
         shader's texture coordinate set (s,t,r) with some mapping of the
         components of a previous texture shader result.  The component
         mapping depends on the type (RGBA or HILO) and signedness of
         the stage's previous texture input.  Other dot product texture
         programs use the result of this program to compose a texture
         coordinate set for a dependent texture access.  The color result
         is undefined.

    15.  DOT_PRODUCT_TEXTURE_2D_NV - When preceded by a DOT_PRODUCT_NV
         program in the previous texture shader stage, computes a second
         similar dot product and composes the two dot products into (s,t)
         texture coordinate set to access a 2D non-projective texture.

    16.  DOT_PRODUCT_TEXTURE_RECTANGLE_NV - Similar to
         DOT_PRODUCT_TEXTURE_2D_NV except that the texture acces is into
         a rectangular non-projective texture.

    17.  DOT_PRODUCT_TEXTURE_CUBE_MAP_NV - When preceded by two
         DOT_PRODUCT_NV programs in the previous two texture shader
         stages, computes a third similar dot product and composes the
         three dot products into (s,t,r) texture coordinate set to access
         a cube map texture.

    18.  DOT_PRODUCT_REFLECT_CUBE_MAP_NV - When preceded by two
         DOT_PRODUCT_NV programs in the previous two texture shader
         stages, computes a third similar dot product and composes the
         three dot products into a normal vector (Nx,Ny,Nz).  An eye
         vector (Ex,Ey,Ez) is composed from the q texture coordinates of
         the three stages.  A reflection vector (Rx,Ry,Rz) is computed
         based on the normal and eye vectors.  The reflection vector
         forms an (s,t,r) texture coordinate set to access a cube map
         texture.

    19.  DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV - Operates like
         DOT_PRODUCT_REFLECT_CUBE_MAP_NV except that the eye vector
         (Ex,Ey,Ez) is a user-defined constant rather than composed from
         the q coordinates of the three stages.

    20.  DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV - When used instead of the second
         DOT_PRODUCT_NV program preceding
         a DOT_PRODUCT_REFLECT_CUBE_MAP_NV or
         DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV stage, the normal
         vector forms an (s,t,r) texture  coordinate set to access a
         cube map texture.
```

<dot product depth replace>

21. DOT_PRODUCT_DEPTH_REPLACE_NV - When preceded by a DOT_PRODUCT_NV program in the previous texture shader stage, computes a second similar dot product and replaces the fragment's window-space depth value with the first dot product results divided by the second. The texture unit RGBA result is (0,0,0,0).

Issues

What should this extension be called? How does the functionality compare with DirectX 8's pixel shaders?

RESOLUTION: This extension is called NV_texture_shader.

DirectX 8 refers to its similar functionality as "pixel shaders". However, DirectX 8 lumps both the functionality described in this extension and additional functionality similar to the functionality in the NV_register_combiners extension together into what DirectX 8 calls pixel shaders. This is confusing in two ways.

1) Pixels are not being shaded. In fact, the DirectX 8 pixel shaders functionality is, taken as a whole, shading only fragments (though Direct3D tends not to make the same clear distinction between fragments and pixels that OpenGL consistently makes).

2) There are two very distinct tasks being performed.

   First, there is the task of interpolated texture coordinate mapping. This per-fragment task maps from interpolated floating-point texture coordinate sets to (typically fixed-point) texture unit RGBA results. In conventional OpenGL, this mapping is performed by accessing the highest priority enabled texture target using the fragment's corresponding interpolated texture coordinate set. This NV_texture_shader extension provides a significantly more powerful mechanism for performing this mapping.

   Second, there is the task of fragment coloring. Fragment coloring is process of combining (typically fixed-point) RGBA colors to generate a final fragment color that, assuming the fragment is not discarded by subsequent per-fragment tests, is used to update the fragment's corresponding pixel in the frame buffer. In conventional OpenGL, fragment coloring is performed by the enabled texture environment functions, fog, and color sum operations. NVIDIA's register combiners functionality (see the NV_register_combiners and NV_register_combiners2 extensions) provides a substantially more powerful alternative to conventional OpenGL fragment coloring.

DirectX 8 has two types of opcodes for pixel shaders. Texture address opcodes correspond to the first task listed above. Texture register opcodes correspond to the second task listed above.

NVIDIA OpenGL extensions maintain a clear distinction between these two tasks. The texture shaders functionality described in this specification corresponds to the first task listed above.

Here is the conceptual framework that NVIDIA OpenGL extensions use to describe shading: Shading is the process of assigning colors to pixels, fragments, or texels. The texture shaders functionality assigns colors to texture unit results (essentially texture shading). These texture unit RGBA results can be used by fragment coloring (fragment shading). The resulting fragments are used to update pixels (pixel shading) possibly via blending and/or multiple rendering passes.

The goal of these individual shading operations is per-pixel shading. Per-pixel shading is accomplished by combining the texture shading, fragment shading, and pixel shading operations, possibly with multiple rendering passes.

Programmable shading is a style of per-pixel shading where the shading operations are expressed in a higher level of abstraction than "raw" OpenGL texture, fragment, and pixel shading operations. In our view, programmable shading does not necessarily require a "pixel program" to be downloaded and executed per-pixel by graphics hardware. Indeed, there are many disadvantages to such an approach in practice. An alternative view of programmable shading (the one that we are promoting) treats the OpenGL primitive shading operations as a SIMD machine and decomposes per-pixel shading programs into one or more OpenGL rendering passes that map to "raw" OpenGL shading operations. We believe that conventional OpenGL combined with NV_register_combiners and NV_texture_shader (and further augmented by programmable geometry via NV_vertex_program and higher-order surfaces via NV_evaluators) can become the hardware basis for a powerful programmable shading system.

The roughly equivalent functionality to DirectX 8's pixel shaders in OpenGL is the combination of NV_texture_shader with NV_register_combiners.

Is anyone working on programmable shading using the NV_texture_shader functionality?

Yes. The Stanford Shading Group is actively working on support for programmable shading using NV_texture_shader, NV_register_combiners, and other extensions as the hardware basis for such a system.

What terms are important to this specification?

texture shaders - A series of texture shader stages that map texture coordinate sets to texture unit RGBA results. An alternative to conventional OpenGL texturing.

texture coordinate set - The interpolated (s,t,r,q) value for a particular texture unit of a particular fragment.

conventional OpenGL texturing - The conventional mechanism used by OpenGL to map texture coordinate sets to texture unit RGBA results whereby a given texture unit's texture coordinate set is used to access the highest priority enabled texture target to generate the texture unit's RGBA result. Conventional OpenGL texturing supports 1D, 2D, 3D, and cube map texture targets. In conventional OpenGL texturing each texture unit operates independently.

texture target type - One of the four texture target types: 1D, 2D, 3D, and cube map. (Note that NV_texture_shader does NOT provide support for 3D textures; the NV_texture_shader2 extension adds texture shader operations for 3D texture targets.)

texture internal format - The internal format of a particular texture object. For example, GL_RGBA8, GL_SIGNED_RGBA8, or GL_SIGNED_HILO16_NV.

texture format type - One of the three texture format types: RGBA, HILO, or texture offset group.

texture component signedness - Whether or not a given component of a texture's texture internal format is signed or not. Signed components are clamped to the range [-1,1] while unsigned components are clamped to the range [0,1].

texture shader enable - The OpenGL enable that determines whether the texture shader functionality (if enabled) or conventional OpenGL texturing functionality (if disabled) is used to map texture coordinate sets to texture unit RGBA results. The enable's initial state is disabled.

texture shader stage - Each texture unit has a corresponding texture shader stage that can be loaded with one of 21 texture shader operations. Depending on the stage's texture shader operation, a texture shader stage uses the texture unit's corresponding texture coordinate set and other state including the texture shader results of previous texture shader stages to generate the stage's particular texture shader result and texture unit RGBA result.

texture unit RGBA result - A (typically fixed-point) color result generated by either a texture shader or conventional OpenGL texturing. This is the color that becomes the texture unit's texture environment function texture input or the initial value of the texture unit's corresponding texture register in the case of register combiners.

texture shader result - The result of a texture shader stage that may be used as an input to a subsequent texture shader stage. This result is distinct from the texture unit RGBA result. The texture shader result may be one of four types: an RGBA color value, a HILO value, a texture offset group value, or a floating-point value. A few texture shader operations are defined to always generate an invalid texture shader result.

texture shader result type - One of the four texture shader result types: RGBA color, HILO, texture offset group, or floating-point.

texture shader operation - One of 21 fixed programs that maps a texture unit's texture coordinate set to a texture shader result and a texture unit RGBA result.

texture consistency - Whether or not the texture object for a given texture target is consistent. The rules for determining consistency depend on the texture target and the texture object's filtering state. For example, a mipmapped texture is inconsistent if its texture levels do not form a consistent mipmap pyramid. Also, a cube map texture is inconsistent if its (filterable) matching cube map faces do not have matching dimensions.

texture shader stage consistency - Whether or not a texture shader stage is consistent or not. The rules for determining texture shader stage consistency depend on the texture shader stage operation and the inputs upon which the texture shader operation depends. For example, texture shader operations that depend on accessing a given texture target are not consistent if the given texture target is not consistent. Also, a texture shader operation that depends on a particular texture shader result type for a previous texture shader result is not consistent if the previous texture shader result type is not appropriate or the previous texture shader stage itself is not consistent. If a texture shader stage is not consistent, it operates as if the operation is the GL_NONE operation.

previous texture input - Some texture shader operations depend on a texture shader result from a specific previous texture input designated by the GL_PREVIOUS_TEXTURE_INPUT_NV state.

What should the default state be?

RESOLUTION: Texture shaders disabled with all stages set to GL_NONE.

How is the mipmap lambda parameter computed for dependent texture fetches?

RESOLUTION: Very carefully. NVIDIA's implementation details are NVIDIA proprietary, but mipmapping of dependent texture fetches is supported.

Does this extension support so-called "bump environment mapping"?

Something similar to DirectX 6 so-called bump environment mapping can be emulated with the GL_OFFSET_TEXTURE_2D_NV texture shader.

A more correct form of bump environment mapping can be implemented by using the following texture shaders:

```
texture unit 0: GL_TEXTURE_2D
texture unit 1: GL_DOT_PRODUCT_NV
texture unit 2: GL_DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV
texture unit 3: GL_DOT_PRODUCT_REFLECT_CUBE_MAP_NV
```

Texture unit 0 should use a normal map for its 2D texture. A GL_SIGNED_RGB texture can encode signed tangent-space normal perturbations. Or for more precision, a GL_SIGNED_HILO_NV texture can encode the normal perturbations in hemisphere fashion.

The tangent (Tx,Ty,Tz), binormal (Bx,By,Bz), and normal (Nx,Ny,Nz) that together map tangent-space normals to cube map-space normals should be sent as texture coordinates s1, t1, r1, s2, t2, r2, s3, t3, and r3 respectively. Typically, cube map space is aligned to match world space.

The (unnormalized) cube map-space eye vector (Ex,Ey,Ez) should be sent as texture coordinates q1, q2, and q3 respectively.

A vertex programs (using the NV_vertex_program extension) can compute and assign the required tangent, binormal, normal, and eye vectors to the appropriate texture coordinates. Conventional OpenGL evaluators (or the NV_evaluators extension) can be used to evaluate the tangent and normal automatically for Bezier patches. The binormal is the cross product of the normal and tangent.

Texture units 1, 2, and 3, should also all specify GL_TEXTURE0_ARB (the texture unit accessing the normal map) for their GL_PREVIOUS_TEXTURE_INPUT_NV parameter.

The three dot product texture shader operations performed by the texture shaders for texture units 1, 2, and 3 form a 3x3 matrix that transforms the tangent-space normal (the result of the texture shader for texture unit 0). This rotates the tangent-space normal into a cube map-space.

Texture unit 2's cube map texture should encode a pre-computed diffuse lighting solution. Texture unit 3's cube map texture should encode a pre-computed specular lighting solution. The specular lighting solution can be an environment map.

Texture unit 2 is accessed using the cube map-space normal vector resulting from the three dot product results of the texture shaders for texture units 1, 2, and 3. (While normally texture shader operations are executed in order, preceding GL_DOT_PRODUCT_REFLECT_CUBE_MAP_NV by GL_DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV is a special case where a dot product result from texture unit 3 influences the cube map access of texture unit 2.)

Texture unit 3 is accessed using the cube map-space reflection vector computed using the cube map-space normal vector from the three dot product results of the texture shaders for texture units 1, 2, and 3 and the cube-map space eye-vector (q1,q2,q3).

Note that using cube maps to access the diffuse and specular illumination obviates the need for an explicit normalization of the typically unnormalized cube map-space normal and reflection vectors.

The register combiners (using the NV_register_combiners extension) can combine the diffuse and specular contribution available in the GL_TEXTURE2_ARB and GL_TEXTURE3_ARB registers respectively. A constant ambient contribution can be stored in a register combiner constant. The ambient contribution could also be folded into the diffuse cube map.

If desired, the diffuse and ambient contribution can be modulated by a diffuse material parameter encoded in the RGB components of the primary color.

If desired, the specular contribution can be modulated by a specular material parameter encoded in the RGB components of the secondary color.

Yes, this is all quite complicated, but the result is a true bump environment mapping technique with excellent accounting for normalization and per-vertex interpolated diffuse and specular materials. An environment and/or an arbitrary number of distant or infinite lights can be encoded into the diffuse and specular cube maps.

Why must GL_DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV be used only in conjunction with GL_DOT_PRODUCT_REFLECT_CUBE_MAP_NV? Why does the GL_DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV stage rely on a result computed in the following stage?

Think of the GL_DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV and GL_DOT_PRODUCT_REFLECT_CUBE_MAP_NV operations as forming a compound operation. The idea is to generate two cube map accesses based on a perturbed normal and reflection vector where the reflection vector is a function of the perturbed normal vector. To minimize the number of stages (three stages only) and reuse the internal computations involved, this is treated as a compound operation.

Note that the GL_DOT_PRODUCT_REFLECT_CUBE_MAP_NV vector can be preceded by two GL_DOT_PRODUCT_NV operations instead of a GL_DOT_PRODUCT_NV operation then a GL_DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV operation. This may be more efficient when only the cube map access using the reflection vector is required (a shiny object without any diffuse reflectance).

Also note that if only the diffuse reflectance cube map access is required, this can be accomplished by simply using the GL_DOT_PRODUCT_CUBE_MAP_NV operation preceded by two GL_DOT_PRODUCT_NV operations.

How do texture shader stages map to register combiner texture registers?

RESOLUTION: If GL_TEXTURE_SHADER_NV is enabled, the texture unit RGBA result for a each texture stage is used to initialize the respective texture register in the register combiners.

So if a texture shader generates a texture unit RGBA result for texture unit 2, use GL_TEXTURE2_ARB for the name of the register value in register combiners.

Should the number of shader stages be settable?

RESOLUTION: No, unused stages can be set to GL_NONE.

How do signed RGBA texture components show up in the register combiners texture registers?

RESOLUTION: As signed values. You can use GL_SIGNED_IDENTITY_NV and get to the signed value directly.

How does the texture unit RGBA result of a GL_NONE, GL_CULL_FRAGMENT_NV, DOT_PRODUCT_NV, or GL_DOT_PRODUCT_DEPTH_REPLACE_NV texture shader operation show up in the register combiners texture registers?

RESOLUTION: Always as the value (0,0,0,0).

How the texture RGBA result of the GL_NONE, GL_CULL_FRAGMENT_NV, GL_DOT_PRODUCT_NV, and GL_DOT_PRODUCT_DEPTH_REPLACE_NV texture shader operations shows up in the texture environment is not an issue, because the texture environment operation is always assumed to be GL_NONE when the corresponding texture shader is one of GL_NONE, GL_CULL_FRAGMENT_NV, GL_DOT_PRODUCT_NV, or GL_DOT_PRODUCT_DEPTH_REPLACE_NV when GL_TEXTURE_SHADER_NV is enabled.

Why introduce new pixel groups (the HILO and texture offset groups)?

RESOLUTION: In core OpenGL, texture image data is transferred and stored as sets of color components. Such color data can always be promoted to RGBA data.

In addition to color components, there are other types of image data in OpenGL including depth components, stencil components, and color indices. Depth and stencil components can be used by glReadPixels, glDrawPixels, and glCopyPixels, but are not useful for storing texture data in core OpenGL. The EXT_paletted_texture and EXT_index_texture extensions extend the contents of textures to include indices (even though in the case of EXT_paletted_texture, texel fetches are always eventually expanded into color components by the texture palette).

However this these existing pixel groups are not sufficient for all the texture shader operations introduced by this extension. Certain texture shader operations require texture data that is not merely a set of color components. The dot product (GL_DOT_PRODUCT_NV, etc) operations both can utilize high-precision hi and lo components. The offset texture operations (GL_OFFSET_TEXTURE_2D_NV, GL_OFFSET_TEXTURE_2D_SCALE_NV, GL_OFFSET_TEXTURE_RECTANGLE_NV, and GL_OFFSET_TEXTURE_RECTANGLE_SCALE_NV) require textures containing signed offsets used to displace texture coordinates. The GL_OFFSET_TEXTURE_2D_SCALE_NV and GL_OFFSET_TEXTURE_RECTANGLE_SCALE_NV also require an unsigned magnitude for the scaling operation.

To facilitate these new texture representations, this extension introduces several new (external) formats, pixel groups, and internal texture formats. An (external) format is the external representation used by an application to specify pixel data for use by OpenGL. A pixel group is a grouping of components that are transformed by OpenGL's pixel transfer mechanism is a particular manner. For example, RGBA components for colors are transformed differently than stencil components when passed through OpenGL's pixel transfer mechanism. An internal texture format is the representation of texture data within OpenGL. Note that the (external) format used to specify the data by the application may be different than the internal texture format used to store the texture data internally to OpenGL. For example, core OpenGL permits an application to specify data for a texture as GL_LUMINANCE_ALPHA data stored in GLfloats even though the data is to be store in a GL_RGBA8 texture. OpenGL's pixel unpacking and pixel transfer operations perform an appropriate transformation of the data when such a texture download is performed. Also note that data from one pixel group (say stencil components) cannot be supplied as data for a different pixel group (say RGBA components).

This extension introduces four new (external) formats for texture data: GL_HILO_NV, GL_DSDT_NV, GL_DSDT_MAG_NV, and GL_DSDT_MAG_VIB_NV.

GL_HILO_NV is for specifying high-precision hi and lo components. The other three formats are used to specify texture offset groups. These new formats can only be used for specifying textures (not copying, reading, or writing pixels).

Each of these four pixel formats belong to one of two pixel groups. Pixels specified with the GL_HILO_NV format are transformed as HILO components. Pixels specified with the DSDT_NV, DSDT_MAG_NV, and DSDT_MAG_VIB_NV formats are transformed as texture offset groups.

The HILO component and texture offset group pixel groups have independent scale and bias operations for each component type. Various pixel transfer operations that are performed on the RGBA components pixel group are NOT performed on these two new pixel groups. OpenGL's pixel map, color table, convolution, color matrix, histogram, and min/max are NOT performed on the HILO components or texture offset group pixel groups.

There are four internal texture formats for texture data specified as HILO components: GL_HILO_NV, GL_HILO16_NV, GL_SIGNED_HILO_NV, and GL_SIGNED_HILO16_NV. The HILO data can be stored as either unsigned [0,1] value or [-1,1] signed values. There are also enumerants for both explicitly sized component precision (16-bit components) and unsized component precision. OpenGL implementations are expected to keep HILO components are high precision even if an unsized internal texture format is used.

The expectation with HILO textures is that applications will specify HILO data using a type of GL_UNSIGNED_SHORT or GL_SHORT or larger data types. Specifying HILO data with GL_UNSIGNED_BYTE or GL_BYTE works but does not exploit the full available precision of the HILO internal texture formats.

There are six internal texture formats for texture data specified as texture offset groups: GL_DSDT_NV, GL_DSDT8_NV, GL_DSDT_MAG_NV, GL_DSDT8_MAG8_NV, GL_DSDT_MAG_INTENSITY_NV and GL_DSDT8_MAG8_INTENSITY8_NV. The GL_DSDT_NV formats specify two signed [-1,1] components, ds and dt, used to offset s and t texture coordinates. The GL_DSDT_MAG_NV formats specify an additional third unsigned [0,1] component that is a magnitude to scale an unsigned RGBA texture fetch by. The GL_DSDT_MAG_INTENSITY_NV formats specify an additional fourth [0,1] unsigned component, intensity, that becomes the intensity of the fetched texture for use in the texture environment or register combiners. There are also enumerants for both explicitly sized (8-bit components) and unsized component precision.

Note that the vibrance (VIB) component of the GL_DSDT_MAG_VIB_NV format becomes the intensity component of the GL_DSDT_MAG_INTENSITY_NV internal texture format. Vibrance becomes intensity in the GL_DSDT_MAG_INTENSITY_NV texture format. The introduction of vibrance is because core OpenGL has no notion of an intensity component in the pixel transfer mechanism or as an external format (instead the red component of an RGBA value becomes the intensity component of intensity textures).

How does the texture unit RGBA result of a texture shader that fetches a texture with a base internal format of GL_HILO_NV, GL_DSDT_NV, or GL_DSDT_MAG_NV show up in the register combiners texture registers?

RESOLUTION: Always as the value (0,0,0,0).

How the texture RGBA result of a texture shader that fetches a texture with a base internal format of GL_HILO_NV, GL_DSDT_NV, or GL_DSDT_MAG_NV the GL_DOT_PRODUCT_NV texture shader shows up in the texture environment is not an issue, because the texture environment operation is always assumed to be GL_NONE in this case when GL_TEXTURE_SHADER_NV is enabled.

Does the GL_DOT_PRODUCT_DEPTH_REPLACE_NV program replace the eye-distance Z or window-space depth?

RESOLUTION: Window-space depth. And if the window-space depth value is outside of the near and far depth range values, the fragment is rejected.

The GL_CULL_FRAGMENT_NV operation always compares against all four texture coordinates. What if I want only one, two, or three comparisons?

RESOLUTION: To compare against a single value, replicate that value in all the coordinates and set the comparison for all components to be identical. Or you can set uninteresting coordinates to zero and use the GL_GEQUAL comparison which will never cull for the value zero.

What is GL_CULL_FRAGMENT_NV good for?

The GL_CULL_FRAGMENT_NV operation provides a mechanism to implement per-fragment clip planes. If a texture coordinate is assigned a signed distance to a plane, the cull fragment test can discard fragments on the wrong side of the plane. Each texture shader stage provides up to four such clip planes. An eye-space clip plane can be established using the GL_EYE_LINEAR texture coordinate generation mode where the clip plane equation is specified via the GL_EYE_PLANE state.

Clip planes are one application for GL_CULL_FRAGMENT_NV, but other clipping approaches are possible too. For example, by computing and assigning appropriate texture coordinates (perhaps with NV_vertex_program), fragments beyond a certain distance from a point can be culled (assuming that it is acceptable to linearly interpolate a distance between vertices).

The texture border color is supposed to be an RGBA value clamped to the range [0,1]. How does the texture border color work in conjunction with signed RGBA color components, HILO components, and texture offset component groups?

RESOLUTION: The per-texture object GL_TEXTURE_BORDER_COLOR is superceded by a GL_TEXTURE_BORDER_VALUES symbolic token. The texture border values are four floats (not clamped to [0,1] when specified). When a texture border is required for a texture, the components for the border texel are determined by the GL_TEXTURE_BORDER_VALUES state. For color components, the GL_TEXTURE_BORDER_VALUES state is treated as a set of RGBA color components. For HILO components, the first value is treated as hi and the second value is treated as lo. For texture offset components, the ds, dt, mag, and vib values correspond to the first, second, third, and fourth texture border values respectively. The particular texture border components are clamped to the range of the component determined by the texture's internal format. So a signed component is clamped to the [-1,1] range and an unsigned component is clamped to the [0,1] range.

For backward compatibility, the GL_TEXTURE_BORDER_COLOR can still be specified and queried. When specified, the values are clamped to [0,1] and used to update the texture border values. When GL_TEXTURE_BORDER_COLOR is queried, there is no clamping of the returned values.

With signed texture components, does the texture environment function discussion need to be amended?

RESOLUTION: Yes. We do not want texture environment results to exceed the range [-1,1].

The GL_DECAL and GL_BLEND operations perform linear interpolations of various components of the form

```
A * B + (1-A) * C
```

The value of A should not be allowed to be negative otherwise, the value of (1-A) may exceed 1.0. These linear interpolations should be written in the form

```
max(0,A) * B + (1-max(0,A)) * C
```

The GL_ADD operation clamps its result to 1.0, but if negative components are permitted, the result should be clamped to the range [-1,1].

The GL_COMBINE_ARB (and GL_COMBINE_EXT) and GL_COMBINE4_NV operations do explicit clamping of all result to [0,1]. In addition, NV_texture_shader adds requirements to clamp inputs to [0,1] too. This is because the GL_ONE_MINUS_SRC_COLOR and GL_ONE_MINUS_SRC_ALPHA operands should really be computing 1-max(0,C). For completeness, GL_SRC_COLOR and GL_SRC_ALPHA should be computing max(0,C).

With signed texture components, does the color sum discussion need to be amended?

RESOLUTION: Yes. The primary and secondary color should both be clamped to the range [0,1] before they are summed.

The unextended OpenGL 1.2 description of color sum does not require a clamp of the primary and secondary colors to the [0,1] range before they are summed. Before signed texture components, the standard texture environment modes either could not generate results outside the [0,1] range or explicitly clamped their results to this range (as in the case of GL_ADD, GL_COMBINE_EXT, and GL_COMBINE4_NV). Now with signed texture components, negative values can be generated by texture environment functions.

We do not want to clamp the intermediate results of texture environment stages since negative results may be useful in subsequent stages, but clamping should be applied to the primary color immediately before the color sum. For symmetry, clamping of the secondary color is specified as well (though there is currently no way to generate a negative secondary color).

Why vibrance?

Vibrance is the fourth component of the external representation of a texture offset group. During pixel transfer, vibrance is scaled and biased based on the GL_VIBRANCE_SCALE and GL_VIBRANCE_BIAS state. Once transformed, the vibrance component becomes the intensity component for textures with a DSDT_MAG_INTENSITY base internal format. Vibrance is meaningful only when specifying texture images with the DS_DT_MAG_VIB_NV external format (and is not supported when reading, drawing, or copying pixels).

There are lots of reasons that a texture shader stage is inconsistent, and in which case, the stage operates as if the operation is NONE. For debugging sanity, is there a way to determine whether a particular texture shader stage is consistent?

RESOLUTION: Yes. Query the shader consistency of a particular texture unit with:

```
  GLint consistent;

  glActiveTextureARB(stage_to_check);
  glGetTexEnviv(GL_TEXTURE_SHADER_NV, GL_SHADER_CONSISTENT_NV,
    &consistent);
```

consistent is one or zero depending on whether the shader stage is consistent or not.

Should there be signed components with sub 8-bit precision?

RESOLUTION: No.

Should packed pixel formats for texture offset groups be supported?

RESOLUTION: Yes, but they are limited to UNSIGNED_INT_S8_S8_8_8_NV and UNSIGNED_INT_8_8_S8_S8_REV_NV for use with the DSDT_MAG_VIB_NV format.

Note that these two new packed pixel formats are only for the DSDT_MAG_VIB_NV and cannot be used with RGBA or BGRA formats. Likewise, the RGBA and BGRA formats cannot be used with the new UNSIGNED_INT_S8_S8_8_8_NV and UNSIGNED_INT_8_8_S8_S8_REV_NV types.

What should be said about signed fixed-point precision and range of actual implementations?

RESOLUTION: The core OpenGL specification typically specifies fixed-point numerical computations without regard to the specific precision of the computations. This practice is intentional because it permits implementations to vary in the degree of precision used for internal OpenGL computations. When mapping unsigned fixed-point values to a [0,1] range, the mapping is straightforward.

However, this extension supports signed texture components in the range [-1,1]. This presents some awkward choices for how to map [-1,1] to a fixed-point representation. Assuming a binary fixed-point representation with an even distribution of precision, there is no way to exactly represent -1, 0, and 1 and avoid representing values outside the [-1,1] range.

This is not a unique issue for this extension. In core OpenGL, table 2.6 describes mappings from unsigned integer types (GLbyte, GLshort, and GLint) that preclude the exact specification of 0.0. NV_register_combiners supports signed fixed-point values that have similar representation issues.

NVIDIA's solution to this representation problem is to use 8-, 9-, and 16-bit fixed-point representations for signed values in the [-1,1] range such that

floating-point	8-bit fixed-point	9-bit fixed-point	16 bit fixed-point
1.0	n/a	255	n/a
0.99996...	n/a	n/a	32767
0.99218...	127	n/a	n/a
0.0	0	0	0
-1.0	-128	-255	-32768
-1.00392...	n/a	-256	n/a

The 8-bit and 16-bit signed fixed-point types are used for signed internal texture formats, while the 9-bit signed fixed-point type is used for register combiners computations.

The 9-bit signed fixed-point type has the disadvantage that a number slightly more negative than -1 can be represented and this particular value is different dependent on the number of bits of fixed-point precision. The advantage of this approach is that 1, 0, and -1 can all be represented exactly.

The 8-bit and 16-bit signed fixed-point types have the disadvantage that 1.0 cannot be exactly represented (though -1.0 and zero can be exactly represented).

The specification however is written using the conventional OpenGL practice (table 2.6) of mapping signed values evenly over the range [-1,1] so that zero cannot be precisely represented. This is done to keep this specification consistent with OpenGL's existing conventions and to avoid the ugliness of specifying a precision-dependent range. We expect leeway in how signed fixed-point values are represented.

The spirit of this extension is that an implicit allowance is made for signed fixed-point representations that cannot exactly represent 1.0.

How should NV_texture_rectangle interact with NV_texture_shader?

NV_texture_rectangle introduces a new texture target similar to GL_TEXTURE_2D but that supports non-power-of-two texture dimensions and several usage restrictions (no mipmapping, etc). Also the imaged texture coordinate range for rectangular textures is [0,width]x[0,height] rather than [0,1]x[0,1].

Four texture shader operations will operate like their 2D texture counter-parts, but will access the rectangular texture target rather than the 2D texture target. These are:

```
GL_TEXTURE_RECTANGLE_NV
GL_OFFSET_TEXTURE_RECTANGLE_NV
GL_OFFSET_TEXTURE_RECTANGLE_SCALE_NV
GL_DOT_PRODUCT_TEXTURE_RECTANGLE_NV
```

A few 2D texture shader operations, namely GL_DEPENDENT_AR_TEXTURE_2D_NV and GL_DEPENDENT_GB_TEXTURE_2D_NV, do not support rectangular textures because turning colors in the [0,1] range into texture coordinates would only access a single corner texel in a rectangular texture. The offset and dot product rectangular texture shader operations support scaling of the dependent texture coordinates so these operations can access the entire image of a rectangular texture. Note however that it is the responsibility of the application to perform the proper scaling.

Note that the 2D and rectangular "offset texture" shaders both use the same matrix, scale, and bias state.

Does the GL_DOT_PRODUCT_DEPTH_REPLACE_NV operation happen before or after polygon offset?

RESOLUTION: After. The window Z (w_z) is computed during rasterization and polygon offset occurs at this point. The depth replace operation occurs after rasterization (at the point that conventional OpenGL calls "texturing") so when the depth value is replaced, the effect of polygon offset (and normal depth interpolation) is lost when using the depth replace operation.

How does the GL_DOT_PRODUCT_DEPTH_REPLACE_NV operation interact with ARB_multisample?

RESOLUTION: The depth value for all covered samples of a multisampled fragment are replaced with the _same_ single depth value computed by the depth replace operation. Without depth replace, the depth values of each sample of a fragment may have slightly different depth values because of the polygon's depth gradient.

How should the clamping work for GL_OFFSET_TEXTURE_2D_SCALE?

RESOLUTION: The scale factor should be clamped to [0,1] prior to scaling red, green, and blue.

Red, green, and blue are guaranteed to be unsigned RGB values so the [0,1] scale factor times the [0,1] RGB values results in [0,1] values so no output clamping need be specified.

New Procedures and Functions

None.

New Tokens

Accepted by the <cap> parameter of Enable, Disable, and IsEnabled, and by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev, and by the <target> parameter of TexEnvf, TexEnvfv, TexEnvi, TexEnviv, GetTexEnvfv, and GetTexEnviv:

TEXTURE_SHADER_NV	0x86DE

When the <target> parameter of TexEnvf, TexEnvfv, TexEnvi, TexEnviv, GetTexEnvfv, and GetTexEnviv is TEXTURE_SHADER_NV, then the value of <pname> may be:

RGBA_UNSIGNED_DOT_PRODUCT_MAPPING_NV	0x86D9
SHADER_OPERATION_NV	0x86DF
OFFSET_TEXTURE_SCALE_NV	0x86E2
OFFSET_TEXTURE_BIAS_NV	0x86E3
OFFSET_TEXTURE_2D_SCALE_NV	alias for OFFSET_TEXTURE_SCALE_NV
OFFSET_TEXTURE_2D_BIAS_NV	deprecated alias for OFFSET_TEXTURE_BIAS_NV
PREVIOUS_TEXTURE_INPUT_NV	0x86E4

When the <target> parameter of TexEnvfv, TexEnviv, GetTexEnvfv, and GetTexEnviv is TEXTURE_SHADER_NV, then the value of <pname> may be:

CULL_MODES_NV	0x86E0
OFFSET_TEXTURE_MATRIX_NV	0x86E1
OFFSET_TEXTURE_2D_MATRIX_NV	deprecated alias for OFFSET_TEXTURE_MATRIX_NV
CONST_EYE_NV	0x86E5

```
    When the <target> parameter GetTexEnvfv and GetTexEnviv is
    TEXTURE_SHADER_NV, then the value of <pname> may be:

        SHADER_CONSISTENT_NV                     0x86DD

    When the <target> and <pname> parameters of TexEnvf, TexEnvfv,
    TexEnvi, and TexEnviv are TEXTURE_ENV and TEXTURE_ENV_MODE
    respectively, then the value of <param> or the value pointed to by
    <params> may be:

        NONE

    When the <target> and <pname> parameters of TexEnvf, TexEnvfv,
    TexEnvi, and TexEnviv are TEXTURE_SHADER_NV and SHADER_OPERATION_NV
    respectively, then the value of <param> or the value pointed to by
    <params> may be:

        NONE

        TEXTURE_1D
        TEXTURE_2D
        TEXTURE_RECTANGLE_NV                     (see NV_texture_rectangle)
        TEXTURE_CUBE_MAP_ARB                     (see ARB_texture_cube_map)

        PASS_THROUGH_NV                          0x86E6
        CULL_FRAGMENT_NV                         0x86E7

        OFFSET_TEXTURE_2D_NV                     0x86E8
        OFFSET_TEXTURE_2D_SCALE_NV               see above, note aliasing
        OFFSET_TEXTURE_RECTANGLE_NV              0x864C
        OFFSET_TEXTURE_RECTANGLE_SCALE_NV        0x864D
        DEPENDENT_AR_TEXTURE_2D_NV               0x86E9
        DEPENDENT_GB_TEXTURE_2D_NV               0x86EA

        DOT_PRODUCT_NV                           0x86EC
        DOT_PRODUCT_DEPTH_REPLACE_NV             0x86ED
        DOT_PRODUCT_TEXTURE_2D_NV                0x86EE
        DOT_PRODUCT_TEXTURE_RECTANGLE_NV         0x864E
        DOT_PRODUCT_TEXTURE_CUBE_MAP_NV          0x86F0
        DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV          0x86F1
        DOT_PRODUCT_REFLECT_CUBE_MAP_NV          0x86F2
        DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV 0x86F3

    When the <target> and <pname> parameters of TexEnvfv and TexEnviv
    are TEXTURE_SHADER_NV and CULL_MODES_NV respectively, then the value
    of <param> or the value pointed to by <params> may be:

        LESS
        GEQUAL

    When the <target> and <pname> parameters of TexEnvf,
    TexEnvfv, TexEnvi, and TexEnviv are TEXTURE_SHADER_NV and
    RGBA_UNSIGNED_DOT_PRODUCT_MAPPING_NV respectively, then the value
    of <param> or the value pointed to by <params> may be:

        UNSIGNED_IDENTITY_NV                     (see NV_register_combiners)
        EXPAND_NORMAL_NV                         (see NV_register_combiners)
    When the <target> and <pname> parameters of TexEnvf,
    TexEnvfv, TexEnvi, and TexEnviv are TEXTURE_SHADER_NV and
    PREVIOUS_TEXTURE_INPUT_NV respectively, then the value of <param>
    or the value pointed to by <params> may be:

        TEXTURE0_ARB
        TEXTURE1_ARB
        TEXTURE2_ARB
        TEXTURE3_ARB
        TEXTURE4_ARB
        TEXTURE5_ARB
        TEXTURE6_ARB
        TEXTURE7_ARB

    Accepted by the <format> parameter of GetTexImage, TexImage1D,
    TexImage2D, TexSubImage1D, and TexSubImage2D:

        HILO_NV                                     0x86F4
        DSDT_NV                                     0x86F5
        DSDT_MAG_NV                                 0x86F6
        DSDT_MAG_VIB_NV                             0x86F7

    Accepted by the <type> parameter of GetTexImage, TexImage1D,
    TexImage2D, TexSubImage1D, and TexSubImage2D:

        UNSIGNED_INT_S8_S8_8_8_NV                   0x86DA
        UNSIGNED_INT_8_8_S8_S8_REV_NV               0x86DB

    Accepted by the <internalformat> parameter of CopyTexImage1D,
    CopyTexImage2D, TexImage1D, and TexImage2D:

        SIGNED_RGBA_NV                              0x86FB
        SIGNED_RGBA8_NV                             0x86FC
        SIGNED_RGB_NV                               0x86FE
        SIGNED_RGB8_NV                              0x86FF
        SIGNED_LUMINANCE_NV                         0x8701
        SIGNED_LUMINANCE8_NV                        0x8702
        SIGNED_LUMINANCE_ALPHA_NV                   0x8703
        SIGNED_LUMINANCE8_ALPHA8_NV                 0x8704
        SIGNED_ALPHA_NV                             0x8705
        SIGNED_ALPHA8_NV                            0x8706
        SIGNED_INTENSITY_NV                         0x8707
        SIGNED_INTENSITY8_NV                        0x8708
        SIGNED_RGB_UNSIGNED_ALPHA_NV                0x870C
        SIGNED_RGB8_UNSIGNED_ALPHA8_NV              0x870D
```

Accepted by the <internalformat> parameter of TexImage1D and TexImage2D:

HILO_NV	
HILO16_NV	0x86F8
SIGNED_HILO_NV	0x86F9
SIGNED_HILO16_NV	0x86FA
DSDT_NV	
DSDT8_NV	0x8709
DSDT_MAG_NV	
DSDT8_MAG8_NV	0x870A
DSDT_MAG_INTENSITY_NV	0x86DC
DSDT8_MAG8_INTENSITY8_NV	0x870B

Accepted by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, GetDoublev, PixelTransferf, and PixelTransferi:

HI_SCALE_NV	0x870E
LO_SCALE_NV	0x870F
DS_SCALE_NV	0x8710
DT_SCALE_NV	0x8711
MAGNITUDE_SCALE_NV	0x8712
VIBRANCE_SCALE_NV	0x8713
HI_BIAS_NV	0x8714
LO_BIAS_NV	0x8715
DS_BIAS_NV	0x8716
DT_BIAS_NV	0x8717
MAGNITUDE_BIAS_NV	0x8718
VIBRANCE_BIAS_NV	0x8719

Accepted by the <pname> parameter of TexParameteriv, TexParameterfv, GetTexParameterfv and GetTexParameteriv:

TEXTURE_BORDER_VALUES_NV	0x871A

Accepted by the <pname> parameter of GetTexLevelParameterfv and GetTexLevelParameteriv:

TEXTURE_HI_SIZE_NV	0x871B
TEXTURE_LO_SIZE_NV	0x871C
TEXTURE_DS_SIZE_NV	0x871D
TEXTURE_DT_SIZE_NV	0x871E
TEXTURE_MAG_SIZE_NV	0x871F

Additions to Chapter 2 of the 1.2 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.2 Specification (Rasterization)

-- Section 3.6 "Pixel Rectangles"

Add four new rows to table 3.2:

Parameter Name	Type	Initial Value	Valid Range
HI_SCALE_NV	float	1.0	(-Inf,+Inf)
LO_SCALE_NV	float	1.0	(-Inf,+Inf)
DS_SCALE_NV	float	1.0	(-Inf,+Inf)
DT_SCALE_NV	float	1.0	(-Inf,+Inf)
MAGNITUDE_SCALE_NV	float	1.0	(-Inf,+Inf)
VIBRANCE_SCALE_NV	float	1.0	(-Inf,+Inf)
HI_BIAS_NV	float	0.0	(-Inf,+Inf)
LO_BIAS_NV	float	0.0	(-Inf,+Inf)
DS_BIAS_NV	float	0.0	(-Inf,+Inf)
DT_BIAS_NV	float	0.0	(-Inf,+Inf)
MAGNITUDE_BIAS_NV	float	0.0	(-Inf,+Inf)
VIBRANCE_BIAS_NV	float	0.0	(-Inf,+Inf)

-- Section 3.6.4 "Rasterization of Pixel Rectangles"

Add before the subsection titled "Unpacking":

"The HILO_NV, DSDT_NV, DSDT_MAG_NV, and DSDT_MAG_VIB_NV formats are described in this section and section 3.6.5 even though these formats are supported only for texture images. Textures with the HILO_NV format are intended for use with certain dot product texture and dependent texture shader operations (see section 3.8.13). Textures with the DSDT_NV, DSDT_MAG_NV, and DSDT_MAG_VIB_NV format are intended for use with certain offset texture 2D texture shader operations (see section 3.8.13).

The error INVALID_ENUM occurs if HILO_NV, DSDT_NV, DSDT_MAG_NV, or DSDT_MAG_VIB_NV is used as the format for DrawPixels, ReadPixels, or other commands that specify or query an image with a format and type parameter though the image is not a texture image. The HILO_NV, DSDT_NV, DSDT_MAG_NV, or DSDT_MAG_VIB_NV formats are intended for use with the TexImage and TexSubImage commands.

The HILO_NV format consists of two components, hi and lo, in the hi then lo order. The hi and lo components maintain at least 16 bits of storage per component (at least 16 bits of magnitude for unsigned components and at least 15 bits of magnitude for signed components).

The DSDT_NV format consists of two signed components ds and dt, in the ds then dt order. The DSDT_MAG_NV format consists of three components: the signed ds and dt components and an unsigned magnitude component (mag for short), in the ds, then dt, then mag order. The DSDT_MAG_VIB_NV format consists of four components: the signed ds and dt components, an unsigned magnitude component (mag for short), and an unsigned vibrance component (vib for short), in the ds, then dt, then mag, then vib order."

```
     Add a new row to table 3.8:

type Parameter                  GL Data  Number of   Matching
Token Name                      Type     Components  Pixel Formats
-----------------------------   -------  ----------  ----------------
UNSIGNED_INT_S8_S8_8_8_NV        uint     4           DSDT_MAG_VIB_NV
UNSIGNED_INT_8_8_S8_S8_REV_NV    uint     4           DSDT_MAG_VIB_NV

     Add to table 3.11:

UNSIGNED_INT_S8_S8_8_8_NV:

 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 1st component         |            2nd        |          3rd          |         4th           |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

UNSIGNED_INT_8_8_S8_S8_REV_NV:

 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|         4th           |            3rd        |          2nd          | 1st component         |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

     Replace the fifth paragraph in the subsection titled "Unpacking"
     with the following:

     "Calling DrawPixels with a type of UNSIGNED_BYTE_3_3_2,
     UNSIGNED_BYTE_2_3_3_REV, UNSIGNED_SHORT_5_6_5,
     UNSIGNED_SHORT_5_6_5_REV, UNSIGNED_SHORT_4_4_4_4,
     UNSIGNED_SHORT_4_4_4_4_REV, UNSIGNED_SHORT_5_5_5_1,
     UNSIGNED_SHORT_1_5_5_5_REV, UNSIGNED_INT_8_8_8_8,
     UNSIGNED_INT_8_8_8_8_REV, UNSIGNED_INT_10_10_10_2, or
     UNSIGNED_INT_2_10_10_10_REV is a special case in which all
     the components of each group are packed into a single unsigned
     byte, unsigned short, or unsigned int, depending on the type.
     When packing or unpacking texture images (for example, using
     TexImage2D or GetTexImage), the type parameter may also be either
     UNSIGNED_INT_S8_S8_8_8_NV or UNSIGNED_INT_8_8_S8_S8_REV though
     neither symbolic token is permitted for DrawPixels, ReadPixels,
     or other commands that specify or query an image with a format
     and type parameter though the image is not a texture image.
     The error INVALID_ENUM occurs when UNSIGNED_INT_S8_S8_8_8_NV is
     used when it is not permitted.  When UNSIGNED_INT_S8_S8_8_8_NV
     or UNSIGNED_INT_8_8_S8_S8_REV_NV is used, the first and second
     components are treated as signed components.  The number of
     components per packed pixel is fixed by the type, and must match the
     number of components per group indicated by the format parameter,
     as listed in table 3.8.  The format must also be one of the formats
     listed in the Matching Pixel Formats column of table 3.8 for the
     specified packed type.  The error INVALID_OPERATION is generated
     if a mismatch occurs.  This constraint also holds for all other
     functions that accept or return pixel data using type and format
     parameters to define the type and format of the data."

     Amend the second sentence in the sixth paragraph in the subsection
     titled "Unpacking" to read:

     "Each bitfield is interpreted as an unsigned integer value unless
```

it has been explicitly been stated that the bitfield contains a signed component. Signed bitfields are treated as two's complement numbers."

Add a new row to table 3.12:

Format	First Component	Second Component	Third Component	Fourth Component
DSDT_MAG_VIB_NV	ds	dt	magnitude	vibrance

Change the last sentence in the first paragraph in the subsection titled "Conversion to floating-point" to read:

"For packed pixel types, each unsigned element in the group is converted by computing c / (2^N-1), where c is the unsigned integer value of the bitfield containing the element and N is the number of bits in the bitfield. In the case of signed elements of a packed pixel type, the signed element is converted by computing 2*c+1 / (2^N-1), where c is the signed integer value of the bitfield containing the element and N is the number of bits in the bitfield."

Change the first sentence in the subsection "Final Expansion to RGBA" to read:

"This step is performed only for groups other than HILO component, depth component, and texture offset groups."

Add the following additional enumeration to the kind of pixel groups in section 3.6.5:

"5. HILO component: Each group comprises two components: hi and lo.

6. Texture offset group: Each group comprises four components: a ds and dt pair, a magnitude, and a vibrance."

Change the subsection "Arithmetic on Components" in section 3.6.5 to read:

"This step applies only to RGBA component, depth component, and HILO component, and texture offset groups. Each component is multiplied by an appropriate signed scale factor: RED_SCALE for an R component, GREEN_SCALE for a G component, BLUE_SCALE for a B component, ALPHA_SCALE, for an A component, HI_SCALE_NV for a HI component, LO_SCALE_NV for a LO component, DS_SCALE_NV for a DS component, DT_SCALE_NV for a DT component, MAGNITUDE_SCALE_NV for a MAG component, VIBRANCE_SCALE_NV for a VIB component, or DEPTH_SCALE for a depth component.

Then the result is added to the appropriate signed bias: RED_BIAS, GREEN_BIAS, BLUE_BIAS, ALPHA_BIAS, HI_BIAS_NV, LO_BIAS_NV, DS_BIAS_NV, DT_BIAS_NV, MAGNITUDE_BIAS_NV, VIBRANCE_BIAS_NV, or DEPTH_BIAS."

-- Section 3.8 "Texturing"

Replace the first paragraph with the following:

"The GL provides two mechanisms for mapping sets of (s,t,r,q) texture coordinates to RGBA colors: conventional texturing and texture shaders.

Conventional texturing maps a portion of a specified image onto each primitive for each enabled texture unit. Conventional texture mapping is accomplished by using the color of an image at the location indicated by a fragment's non-homogeneous (s,t,r) coordinates for a given texture unit.

The alternative to conventional texturing is the texture shaders mechanism. When texture shaders are enabled, each texture unit uses one of twenty-one texture shader operations. Eighteen of the twenty-one shader operations map an (s,t,r,q) texture coordinate set to an RGBA color. Of these, three texture shader operations directly correspond to the 1D, 2D, and cube map conventional texturing operations. Depending on the texture shader operation, the mapping from the (s,t,r,q) texture coordinate set to an RGBA color may depend on the given texture unit's currently bound texture object state and/or the results of previous texture shader operations. The three remaining texture shader operations respectively provide a fragment culling mechanism based on texture coordinates, a means to replace the fragment depth value, and a dot product operation that computes a floating-point value for use by subsequent texture shaders. The specifics of each texture shader operation are described in section 3.8.12.

Texture shading is enabled or disabled using the generic Enable and Disable commands, respectively, with the symbolic constant TEXTURE_SHADER_NV. When texture shading is disabled, conventional texturing generates an RGBA color for each enabled textures unit as described in Sections 3.8.10.

After RGBA colors are assigned to each texture unit, either by conventional texturing or texture shaders, the GL proceeds with fragment coloring, either using the texture environment, fog, and color sum operations, or using register combiners extension if supported.

Neither conventional texturing nor texture shaders affects the secondary color."

-- Section 3.8.1 "Texture Image Specification"

Add the following sentence to the first paragraph:

"The formats HILO_NV, DSDT_NV, DSDT_MAG_NV, and DSDT_MAG_VIB_NV are allowed for specifying texture images."

Replace the fourth paragraph with:

"The selected groups are processed exactly as for DrawPixels, stopping just before conversion. Each R, G, B, A, HI, LO, DS, DT,

    and MAG value so generated is clamped to [0,1] if the corresponding
    component is unsigned, or if the corresponding component is signed,
    is clamped to [-1,1]. The signedness of components depends on the
    internal format (see table 3.16). The signedness of components
    for unsized internal formats matches the signedness of components
    for any respective sized version of the internal format."

    Replace table 3.15 with the following table:

Base Internal Format	Component Values	Internal Components	Format Type
ALPHA	A	A	RGBA
LUMINANCE	R	L	RGBA
LUMINANCE_ALPHA	R,A	L,A	RGBA
INTENSITY	R	I	RGBA
RGB	R,G,B	R,G,B	RGBA
RGBA	R,G,B,A	R,G,B,A	RGBA
HILO_NV	HI,LO	HI,LO	HILO
DSDT_NV	DS,DT	DS,DT	texture offset group
DSDT_MAG_NV	DS,DT,MAG	DS,DT,MAG	texture offset group
DSDT_MAG_INTENSITY_NV	DS,DT,MAG,VIB	DS,DT,MAG,I	RGBA/texture offset group

    Re-caption table 3.15 as:

    "Conversion from RGBA, HILO, and texture offset pixel components to
    internal texture table, or filter components. See section 3.8.9
    for a description of the texture components R, G, B, A, L, and I.
    See section 3.8.13 for an explanation of the handling of the texture
    components HI, LO, DS, DT, MAG, and VIB."

    Add five more columns to table 3.16 labeled "HI bits", "LO bits", "DS
    bits", "DT bits", and "MAG bits". Existing table rows should have
    these column entries blank. Add the following rows to the table:

Sized Internal Format	Base Internal Format	R bits	G bits	B bits	A bits	L bits	I bits	HI bits	LO bits	DS bits	DT bits	MAG bits
HILO16_NV	HILO							16	16			
SIGNED_HILO16_NV	HILO							16*	16*			
SIGNED_RGBA8_NV	RGBA	8*	8*	8*	8*							
SIGNED_RGB8_UNSIGNED_ALPHA8_NV	RGBA	8*	8*	8*	8							
SIGNED_RGB8_NV	RGB	8*	8*	8*								
SIGNED_LUMINANCE8_NV	LUMINANCE					8*						
SIGNED_LUMINANCE8_ALPHA8_NV	LUMINANCE_ALPHA				8*	8*						
SIGNED_ALPHA8_NV	ALPHA				8*							
SIGNED_INTENSITY8_NV	INTENSITY						8*					
DSDT8_NV	DSDT_NV									8*	8*	
DSDT8_MAG8_NV	DSDT_MAG_NV									8*	8*	8
DSDT8_MAG8_INTENSITY8_NV	DSDT_MAG_INTENSITY_NV						8			8*	8*	8

    Add to the caption for table 3.16:

    "An asterisk (*) following a component size indicates that the
    corresponding component is signed (the sign bit is included in
    specified component resolution size)."

    Change the first sentences of the fifth paragraph to read:

    "Components are then selected from the resulting R, G, B, A, HI, LO,
    DS, DT, and MAG values to obtain a texture with the base internal
    format specified by (or derived from) internalformat. Table 3.15
    summarizes the mapping of R, G, B, A, HI, LO, DS, DT, and MAG values

to texture components, as a function of the base internal format of the texture image. internalformat may be specified as one of the ten base internal format symbolic constants listed in table 3.15, or as one of the sized internal format symbolic constants listed in table 3.16."

Add these sentences before the last sentence in the fifth paragraph:

"The error INVALID_OPERATION is generated if the format is HILO_NV and the internalformat is not one of HILO_NV, HILO16_NV, SIGNED_HILO_NV, SIGNED_HILO16_NV; or if the internalformat is one of HILO_NV, HILO16_NV, SIGNED_HILO_NV, or SIGNED_HILO16_NV and the format is not HILO_NV.

The error INVALID_OPERATION is generated if the format is DSDT_NV and the internalformat is not either DSDT_NV or DSDT8_NV; or if the internal format is either DSDT_NV or DSDT8_NV and the format is not DSDT_NV.

The error INVALID_OPERATION is generated if the format is DSDT_MAG_NV and the internalformat is not either DSDT_MAG_NV or DSDT8_MAG8_NV; or if the internal format is either DSDT_MAG_NV or DSDT8_MAG8_NV and the format is not DSDT_MAG_NV.

The error INVALID_OPERATION is generated if the format is DSDT_MAG_VIB_NV and the internalformat is not either DSDT_MAG_INTENSITY_NV or DSDT8_MAG8_INTENSITY8_NV; or if the internal format is either DSDT_MAG_INTENSITY_NV or DSDT8_MAG8_INTENSITY8_NV and the format is not DSDT_MAG_VIB_NV."

Change the first sentence of the sixth paragraph to read:

"The internal component resolution is the number of bits allocated to each value in a texture image (and includes the sign bit if the component is signed)."

Change the third sentence of the sixth paragraph to read:

"If a sized internal format is specified, the mapping of the R, G, B, A, HI, LO, DS, DT, and MAG values to texture components is equivalent to the mapping of the corresponding base internal format's components, as specified in table 3.15, and the memory allocations per texture component is assigned by the GL to match the allocations listed in table 3.16 as closely as possible."

-- Section 3.8.2 "Alternate Texture Image Specification Commands"

In the second paragraph (describing CopyTexImage2D), change the third to the last sentence to:

"Parameters level, internalformat, and border are specified using the same values, with the same meanings, as the equivalent arguments of TexImage2D, except that internalformat may not be specified as 1, 2, 3, 4, HILO_NV, HILO16_NV, SIGNED_HILO_NV, SIGNED_HILO16_NV, DSDT_NV, DSDT8_NV, DSDT_MAG_NV, DSDT8_MAG8_NV, DSDT_MAG_INTENSITY_NV, or DSDT8_MAG8_INTENSITY8_NV."

    In the third paragraph (describing CopyTexImage1D), change the
    second to the last sentence to:

    "level, internalformat, and border are specified using the same
    values, with the same meanings, as the equivalent arguments of
    TexImage1D, except that internalformat may not be specified as 1, 2,
    3, 4, HILO_NV, HILO16_NV, SIGNED_HILO_NV, SIGNED_HILO16_NV, DSDT_NV,
    DSDT8_NV, DSDT_MAG_NV, DSDT8_MAG8_NV, DSDT_MAG_INTENSITY_NV, or
    DSDT8_MAG8_INTENSITY8_NV."

    Insert the following text after the six paragraph reading:

    "CopyTexSubImage2D and CopyTexSubImage1D generate the error
    INVALID_OPERATION if the internal format of the texture array to
    which the pixels are to be copied is one of HILO_NV, HILO16_NV,
    SIGNED_HILO_NV, SIGNED_HILO16_NV, DSDT_NV, DSDT8_NV, DSDT_MAG_NV,
    DSDT8_MAG8_NV, DSDT_MAG_INTENSITY_NV, or DSDT8_MAG8_INTENSITY8_NV.

    TexSubImage2D and TexSubImage1D generate the error INVALID_OPERATION
    if the internal format of the texture array to which the texels are
    to be copied has a different format type (according to table 3.15)
    than the format type of the texels being specified. Specifically, if
    the base internal format is not one of HILO_NV, DSDT_NV, DSDT_MAG_NV,
    or DSDT_INTENSITY_NV, then the format parameter must be one of
    COLOR_INDEX, RED, GREEN, BLUE, ALPHA, RGB, RGBA, LUMINANCE, or
    LUMINANCE_ALPHA; if the base internal format is HILO_NV, then the
    format parameter must be HILO_NV; if the base internal format is
    DSDT_NV, then the format parameter must be DSDT_NV; if the base
    internal format is DSDT_MAG_NV, then the format parameter must be
    DSDT_MAG_NV; if the base internal format is DSDT_MAG_INTENSITY_NV,
    the format parameter must be DSDT_MAG_VIB_NV."

-- Section 3.8.3 "Texture Parameters"

    Change the TEXTURE_BORDER_COLOR line in table 3.17 to read:

Name	Type	Legal Values
TEXTURE_BORDER_VALUES	4 floats	any value

    Add the last two sentences to read:

    "The TEXTURE_BORDER_VALUES state can also be specified with the
    TEXTURE_BORDER_COLOR symbolic constant. When the state is specified
    via TEXTURE_BORDER_COLOR, each of the four values specified are
    first clamped to lie in [0,1]. However, if the texture border
    values state is specified using TEXTURE_BORDER_VALUES, no clamping
    occurs. In either case, if the values are specified as integers,
    the conversion for signed integers from table 2.6 is applied to
    convert the values to floating-point."

-- Section 3.8.5 "Texture Minification"

    Change the last paragraph to read:

    "If any of the selected tauijk, tauij, or taui in the above equations
    refer to a border texel with i < -bs, j < bs, k < -bs, i >= ws-bs, j

>= hs-bs, or k >= ds-bs, then the border values given by the current setting of TEXTURE_BORDER_VALUES is used instead of the unspecified value or values. If the texture contains color components, the components of the TEXTURE_BORDER_VALUES vector are interpreted as an RGBA color to match the texture's internal format in a manner consistent with table 3.15. If the texture contains HILO components, the first and second components of the TEXTURE_BORDER_VALUES vector are interpreted as the hi and lo components respectively. If the texture contains texture offset group components, the first, second, third, and fourth components of the TEXTURE_BORDER_VALUES vector are interpreted as ds, dt, mag, and vib components respectively. Additionally, the texture border values are clamped appropriately depending on the signedness of each particular component. Unsigned components are clamped to [0,1]; signed components are clamped to [-1,1]."

-- Section 3.8.9 "Texture Environment and Texture Functions"

Augment the list of supported texture functions in the first paragraph to read:

"TEXTURE_ENV_MODE may be set to one of REPLACE, MODULATE, DECAL, BLEND, ADD, COMBINE_ARB (or COMBINE_EXT), COMBINE4_NV, or NONE;"

Insert this paragraph between the first and second paragraphs:

"When texture shaders are enabled (see section 3.8.13), a given texture unit's texture shader result may be intended for use as an input to another texture shader stage rather than generating a texture unit RGBA result for use in the given texture unit's texture environment function. Additionally, several texture shader operations and texture format types are intended only to generate texture shader results for subsequent texture shaders or perform a side effect (such as culling the fragment or replacing the fragment's depth value) rather than supplying a useful texture unit RGBA result for use in the texture environment function. For this reason, the NONE texture environment ignores the texture unit RGBA result and passes through its input fragment color unchanged."

Change the third sentence of the second paragraph to read:

"If the TEXTURE_SHADER_NV mode is disabled, the precise form of the texture environment function depends on the base internal format of the texture object bound to the given texture unit's highest-precedence enabled texture target. Otherwise if the TEXTURE_SHADER_NV mode is enabled, then the form of the function depends on the texture unit's texture shader operation.

If a texture shader operation requires fetching a filtered texture color value (though not a HILO or texture offset value; see the subsequent HILO and texture offset discussion), the texture environment function depends on the base internal format of the texture shader operation's respective texture target used for fetching by the texture shader operation.

The PASS_THROUGH_NV texture shader operation does not fetch from any texture target, but it generates an RGBA color and therefore always operates as if the base internal format is RGBA for determining what texture environment function to apply.

If the TEXTURE_SHADER_NV mode is enabled and the texture shader operation for a given texture unit is one of NONE, CULL_FRAGMENT_NV, DOT_PRODUCT_NV, or DOT_PRODUCT_DEPTH_REPLACE_NV, then the given texture unit's texture function always operates as if the texture function is NONE.

If the base internal format of the texture is HILO_NV, DSDT_NV, or DSDT_MAG_NV (independent of whether or not the TEXTURE_SHADER_NV mode is enabled or disabled), then corresponding the texture function always operates as if the texture function is NONE.

If the base internal format of the texture is DSDT_MAG_INTENSITY_NV (independent of whether or not the TEXTURE_SHADER_NV mode is enabled or disabled), then the corresponding texture function operates as if the base internal format is INTENSITY for the purposes of determining the appropriate function using the vibrance component as the intensity value."

Change the phrase in the fourth sentence of the second paragraph describing how Rt, Gt, Bt, At, Lt, and It are assigned to:

"when TEXTURE_SHADER_NV is disabled, Rt, Gt, Bt, At, Lt, and It are the filtered texture values; when TEXTURE_SHADER_NV is enabled, Rt, Gt, Bt, and At are the respective components of the texture unit RGBA result of the texture unit's texture shader stage, and Lt and It are any red, green, or blue component of the texture unit RGBA result (the three components should be the same);"

Change the second to last sentence of the second paragraph to read:

"The initial primary color and texture environment color component values are in the range [0,1]. The filtered texture color and texture function result color component values are in the range [-1,1]. Negative filtered texture color component values are generated by texture internal formats with signed components such as SIGNED_RGBA."

Also amend tables 3.18 and 3.19 based on the following updated columns:

Base Internal Format	DECAL Texture Function	BLEND Texture Function	ADD Texture Function
ALPHA	Rv = Rf (no longer undefined)	Rv = Rf	Rv = Rf
	Gv = Gf	Gv = Gf	Gv = Gf
	Bv = Bf	Bv = Bf	Bv = Rf
	Av = Af	Av = Af*At	Av = Af*Av = At
LUMINANCE (or 1)	Rv = Rf (no longer undefined)	Rv = Rf*(1-max(0,Lt)) + Rc*max(0,Lt)	Rv = max(-1,min(1,Rf+Lt))
	Gv = Gf	Gv = Gf*(1-max(0,Lt)) + Gc*max(0,Lt)	Gv = max(-1,min(1,Gf+Lt))
	Bv = Bf	Bv = Bf*(1-max(0,Lt)) + Bc*max(0,Lt)	Bv = max(-1,min(1,Bf+Lt))
	Av = Af	Av = Af	Av = Af
LUMINANCE_ALPHA (or 2)	Rv = Rf (no longer undefined)	Rv = Rf*(1-max(0,Lt)) + Rc*max(0,Lt)	Rv = max(-1,min(1,Rf+Lt))
	Gv = Gf	Gv = Gf*(1-max(0,Lt)) + Gc*max(0,Lt)	Gv = max(-1,min(1,Gf+Lt))
	Bv = Bf	Bv = Bf*(1-max(0,Lt)) + Bc*max(0,Lt)	Bv = max(-1,min(1,Bf+Lt))
	Av = Af	Av = Af*At	Av = Af*At
INTENSITY	Rv = Rf (no longer undefined)	Rv = Rf*(1-max(0,It)) + Rc*max(0,It)	Rv = max(-1,min(1,Rf+It))
	Gv = Gf	Gv = Gf*(1-max(0,It)) + Gc*max(0,It)	Gv = max(-1,min(1,Gf+It))
	Bv = Bf	Bv = Bf*(1-max(0,It)) + Bc*max(0,It)	Bv = max(-1,min(1,Bf+It))
	Av = Af	Av = Af*(1-max(0,It)) + Ac*max(0,It)	Av = max(-1,min(1,Af+It))
RGB (or 3)	Rv = Rt	Rv = Rf*(1-max(0,Rt)) + Rc*max(0,Rt)	Rv = max(-1,min(1,Rf+Rt))
	Gv = Gt	Gv = Gf*(1-max(0,Gt)) + Gc*max(0,Gt)	Gv = max(-1,min(1,Gf+Gt))
	Bv = Bt	Bv = Bf*(1-max(0,Bt)) + Bc*max(0,Bt)	Bv = max(-1,min(1,Bf+Bt))
	Av = Af	Av = Af	Av = Af
RGBA (or 4)	Rv = Rf*(1-max(0,At)) + Rt*max(0,At)	Rv = Rf*(1-max(0,Rt)) + Rc*max(0,Rt)	Rv = max(-1,min(1,Rf+Rt))
	Gv = Gf*(1-max(0,At)) + Gt*max(0,At)	Gv = Gf*(1-max(0,Gt)) + Gc*max(0,Gt)	Gv = max(-1,min(1,Gf+Gt))
	Bv = Bf*(1-max(0,At)) + Bt*max(0,At)	Bv = Bf*(1-max(0,Bt)) + Bc*max(0,Bt)	Bv = max(-1,min(1,Bf+Bt))
	Av = Af	Av = Af*At	Av = Af*At

Also augment table 3.18 or 3.19 with the following column:

Base Internal Format	NONE Texture Function
ALPHA	Rv = Rf
	Gv = Gf
	Bv = Bf
	Av = Af
LUMINANCE	Rv = Rf
(or 1)	Gv = Gf
	Bv = Bf
	Av = Af
LUMINANCE_ALPHA	Rv = Rf
(or 2)	Gv = Gf
	Bv = Bf
	Av = Af
INTENSITY	Rv = Rf
	Gv = Gf
	Bv = Bf
	Av = Af
RGB	Rv = Rf
(or 3)	Gv = Gf
	Bv = Bf
	Av = Af
RGBA	Rv = Rf
(or 4)	Gv = Gf
	Bv = Bf
	Av = Af

Amend tables 3.21 and 3.22 in the ARB_texture_env_combine specification (or EXT_texture_env_combine specification) to require inputs to be clamped positive (the TEXTURE<n>_ARB entries apply only if NV_texture_env_combine4 is supported):

SOURCE<n>_RGB_EXT	OPERAND<n>_RGB_EXT	Argument
TEXTURE	SRC_COLOR	max(0,Ct)
	ONE_MINUS_SRC_COLOR	(1-max(0,Ct))
	SRC_ALPHA	max(0,At)
	ONE_MINUS_SRC_ALPHA	(1-max(0,At))
CONSTANT_EXT	SRC_COLOR	max(0,Cc
	ONE_MINUS_SRC_COLOR	(1-max(0,Cc)
	SRC_ALPHA	max(0,Ac
	ONE_MINUS_SRC_ALPHA	(1-max(0,Ac)
PRIMARY_COLOR_EXT	SRC_COLOR	max(0,Cf
	ONE_MINUS_SRC_COLOR	(1-max(0,Cf)
	SRC_ALPHA	max(0,Af
	ONE_MINUS_SRC_ALPHA	(1-max(0,Af)
PREVIOUS_EXT	SRC_COLOR	max(0,Cp
	ONE_MINUS_SRC_COLOR	(1-max(0,Cp)
	SRC_ALPHA	max(0,Ap
	ONE_MINUS_SRC_ALPHA	(1-max(0,Ap)
TEXTURE<n>_ARB	SRC_COLOR	max(0,Ct<n>)
	ONE_MINUS_SRC_COLOR	(1-max(0,Ct<n>))
	SRC_ALPHA	max(0,At<n>)
	ONE_MINUS_SRC_ALPHA	(1-max(0,At<n>))

Table 3.21: Arguments for COMBINE_RGB_ARB (or COMBINE_RGB_EXT) functions

SOURCE<n>_ALPHA_EXT	OPERAND<n>_ALPHA_EXT	Argument
TEXTURE	SRC_ALPHA	max(0,At)
	ONE_MINUS_SRC_ALPHA	(1-max(0,At))
CONSTANT_EXT	SRC_ALPHA	max(0,Ac)
	ONE_MINUS_SRC_ALPHA	(1-max(0,Ac))
PRIMARY_COLOR_EXT	SRC_ALPHA	max(0,Af)
	ONE_MINUS_SRC_ALPHA	(1-max(0,Af))
PREVIOUS_EXT	SRC_ALPHA	max(0,Ap)
	ONE_MINUS_SRC_ALPHA	(1-max(0,Ap))
TEXTURE<n>_ARB	SRC_ALPHA	max(0,At<n>)
	ONE_MINUS_SRC_ALPHA	(1-max(0,At<n>))

Table 3.22: Arguments for COMBINE_ALPHA_ARB (or COMBINE_ALPHA_EXT) functions

-- Section 3.9 "Color Sum"

Update the first paragraph to read:

"At the beginning of color sum, a fragment has two RGBA colors: a primary color cpri (which texturing, if enabled, may have modified) and a secondary color csec. The components of these two colors are clamped to [0,1] and then summed to produce a single post-texturing RGBA color c. The components of c are then clamped to the range [0,1]."

-- NEW Section 3.8.13 "Texture Shaders"

"Each texture unit is configured with one of twenty-one texture shader operations. Several texture shader operations require additional state. All per-texture shader stage state is specified using the TexEnv commands with the target specified as TEXTURE_SHADER_NV. The per-texture shader state is replicated per texture unit so the texture unit selected by ActiveTextureARB determines which texture unit's environment is modified by TexEnv calls.

When calling TexEnv with a target of TEXTURE_SHADER_NV, pname must be one of SHADER_OPERATION_NV, CULL_MODES_NV, OFFSET_TEXTURE_MATRIX_NV, OFFSET_TEXTURE_SCALE_NV, OFFSET_TEXTURE_BIAS_NV, PREVIOUS_TEXTURE_INPUT_NV, or CONST_EYE_NV.

When TexEnv is called with the target of TEXTURE_SHADER_NV, SHADER_OPERATION_NV may be set to one of NONE, TEXTURE_1D, TEXTURE_2D, TEXTURE_CUBE_MAP_ARB, PASS_THROUGH_NV, CULL_FRAGMENT_NV, OFFSET_TEXTURE_2D_NV, OFFSET_TEXTURE_2D_SCALE_NV, OFFSET_TEXTURE_RECTANGLE_NV, OFFSET_TEXTURE_RECTANGLE_SCALE_NV, DEPENDENT_AR_TEXTURE_2D_NV, DEPENDENT_GB_TEXTURE_2D_NV, DOT_PRODUCT_NV, DOT_PRODUCT_DEPTH_REPLACE_NV, DOT_PRODUCT_TEXTURE_2D_NV, DOT_PRODUCT_TEXTURE_RECTANGLE_NV, DOT_PRODUCT_TEXTURE_CUBE_MAP_NV, DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV, DOT_PRODUCT_REFLECT_CUBE_MAP_NV, or DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV. The semantics of each of these shader operations is described in section 3.8.13.1. Not every operation is supported in every texture unit. The restrictions for how these shader operations can be configured in various texture units are described in section 3.8.13.2.

When TexEnv is called with the target of TEXTURE_SHADER_NV, CULL_MODES_NV is set to a vector of four cull comparisons by providing four symbolic tokens, each being either LESS or GEQUAL. These cull modes are used by the CULL_FRAGMENT_NV operation (see section 3.8.13.1.7).

When TexEnv is called with the target of TEXTURE_SHADER_NV, RGBA_UNSIGNED_DOT_PRODUCT_MAPPING_NV may be set to either UNSIGNED_IDENTITY_NV or EXPAND_NORMAL_NV. This RGBA unsigned dot product mapping mode is used by the DOT_PRODUCT_NV operation (see section 3.8.13.1.14) and other operations that compute dot products.

When TexEnv is called with the target of TEXTURE_SHADER_NV, PREVIOUS_TEXTURE_INPUT_NV may be set to TEXTUREi_ARB where i is between 0 and n-1 where n is the implementation-dependent number of texture units supported. The INVALID_OPERATION error is generated if i is greater than or equal to the current active texture unit.

When TexEnv is called with the target of TEXTURE_SHADER_NV, OFFSET_TEXTURE_MATRIX_NV may be set to a 2x2 matrix of floating-point values stored in column-major order as 4 consecutive floating-point values, i.e. as:

```
[ a1 a3 ]
[ a2 a4 ]
```

This matrix is used by the OFFSET_TEXTURE_2D_NV, OFFSET_TEXTURE_2D_SCALE_NV, OFFSET_TEXTURE_RECTANGLE_NV, and OFFSET_TEXTURE_RECTANGLE_SCALE_NV operations (see sections 3.8.13.1.8 through 3.8.13.1.11).

When TexEnv is called with the target of TEXTURE_SHADER_NV, OFFSET_TEXTURE_SCALE_NV may be set to a floating-point value. When TexEnv is called with the target of TEXTURE_SHADER_NV, OFFSET_TEXTURE_BIAS_NV may be set to a floating-point value. These scale and bias values are used by the OFFSET_TEXTURE_2D_SCALE_NV and OFFSET_TEXTURE_RECTANGLE_SCALE_NV operations (see section 3.8.13.1.9 and 3.8.13.1.11).

When TexEnv is called with the target of TEXTURE_SHADER_NV, CONST_EYE_NV is set to a vector of three floating-point values used as the constant eye vector in the DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV texture shader (see section 3.8.13.1.19).

3.8.13.1 Texture Shader Operations

The texture enables described in section 3.8.10 only affect conventional texturing mode; these enables are ignored when TEXTURE_SHADER_NV is enabled. Instead, the texture shader operation determines how texture coordinates are mapped to filtered texture values.

Tables 3.A, 3.B, 3.C, and 3.D specify inter-stage dependencies, texture target dependencies, relevant inputs, and result types and values respectively for each texture shader operation. Table 3.E specifies how the components of an accessed texture are mapped to the components of the texture unit RGBA result based on the base internal format of the accessed texture. The following discussion describes each possible texture shader operation in detail.

texture shader				
texture shader operation i	previous texture input	texture shader operation i-1	operation i-2	texture shader operation i+1
NONE	-	-	-	-
TEXTURE_1D	-	-	-	-
TEXTURE_2D	-	-	-	-
TEXTURE_RECTANGLE_NV	-	-	-	-
TEXTURE_CUBE_MAP_ARB	-	-	-	-
PASS_THROUGH_NV	-	-	-	-
CULL_FRAGMENT_NV	-	-	-	-
OFFSET_TEXTURE_2D_NV	base internal texture format must be one of DSDT_NV, DSDT_MAG_NV, or DSDT_MAG_INTENSITY_NV	-	-	-
OFFSET_TEXTURE_2D_SCALE_NV	base internal texture format must be either DSDT_MAG_NV or DSDT_MAG_INTENSITY_NV	-	-	-
OFFSET_TEXTURE_RECTANGLE_NV	base internal texture format must be one of DSDT_NV, DSDT_MAG_NV, or DSDT_MAG_INTENSITY_NV	-	-	-
OFFSET_TEXTURE_RECTANGLE_SCALE_NV	base internal texture format must be either DSDT_MAG_NV or DSDT_MAG_INTENSITY_NV	-	-	-
DEPENDENT_AR_TEXTURE_2D_NV	shader result type must all be unsigned RGBA	-	-	-
DEPENDENT_GB_TEXTURE_2D_NV	shader result type must all be unsigned RGBA	-	-	-
DOT_PRODUCT_NV	shader result type must be one of signed HILO, unsigned HILO, all signed RGBA, or all unsigned RGBA	-	-	-
DOT_PRODUCT_TEXTURE_2D_NV	shader result type must be one of signed HILO, unsigned HILO, all signed RGBA, or all unsigned RGBA	shader operation must be DOT_PRODUCT_NV	-	-
DOT_PRODUCT_TEXTURE_RECTANGLE_NV	shader result type must be one of signed HILO, unsigned HILO, all signed RGBA, all unsigned RGBA	shader operation must be DOT_PRODUCT_NV	-	-
DOT_PRODUCT_TEXTURE_CUBE_MAP_NV	shader result type must be one of signed HILO, unsigned HILO, all signed RGBA, or all unsigned RGBA	shader operation must be DOT_PRODUCT_NV	shader operation must be DOT_PRODUCT_NV	-
DOT_PRODUCT_REFLECT_CUBE_MAP_NV	shader result type must be one of signed HILO, unsigned HILO, all signed RGBA, or all unsigned RGBA; previous texture input must not be unit i-1	shader operation must be DOT_PRODUCT_NV or DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV	shader operation must be DOT_PRODUCT_NV	-
DOT_PRODUCT_CONST_EYE_-REFLECT_CUBE_MAP_NV	shader result type must be one of signed HILO, unsigned HILO, all signed RGBA, or all unsigned RGBA	shader operation must be DOT_PRODUCT_NV or DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV	shader operation must be DOT_PRODUCT_NV	-
DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV	shader result type must be one of signed HILO, unsigned HILO, all signed RGBA, or all unsigned RGBA	shader operation must be DOT_PRODUCT_NV	-	shader operation must be DOT_PRODUCT_REFLECT_CUBE_MAP_NV or DOT_PRODUCT_CONST_EYE_-REFLECT_CUBE_MAP_NV
DOT_PRODUCT_DEPTH_REPLACE_NV	shader result type must be one of signed HILO, unsigned HILO, all signed RGBA, or all unsigned RGBA	shader operation must be DOT_PRODUCT_NV	-	-

Table 3.A: Texture shader inter-stage dependencies for each operation. If any one of the dependencies listed above is not met, the texture shader stage is considered inconsistent. Further texture shader target dependencies are listed in table X.Y. Additionally, if any one of the texture shader stages that a particular texture shader stage depends on is inconsistent, then the dependent texture shader stage is also considered inconsistent. When a texture shader stage is considered inconsistent, the inconsistent stage operates as if the stage's operation is NONE.

texture shader operation i	texture unit i
NONE	-
TEXTURE_1D	1D target must be consistent
TEXTURE_2D	2D target must be consistent
TEXTURE_RECTANGLE_NV	rectangle target must be consistent
TEXTURE_CUBE_MAP_ARB	cube map target must be consistent
PASS_THROUGH_NV	-
CULL_FRAGMENT_NV	-
OFFSET_TEXTURE_2D_NV	2D target must be consistent
OFFSET_TEXTURE_2D_SCALE_NV	2D target must be consistent and 2D texture target type must be unsigned RGBA
OFFSET_TEXTURE_RECTANGLE_NV	rectangle target must be consistent
OFFSET_TEXTURE_RECTANGLE_SCALE_NV	rectangle target must be consistent and rectangle texture target type must be unsigned RGBA
DEPENDENT_AR_TEXTURE_2D_NV	2D target must be consistent
DEPENDENT_GB_TEXTURE_2D_NV	2D target must be consistent
DOT_PRODUCT_NV	-
DOT_PRODUCT_TEXTURE_2D_NV	2D target must be consistent
DOT_PRODUCT_TEXTURE_RECTANGLE_NV	rectangle target must be consistent
DOT_PRODUCT_TEXTURE_CUBE_MAP_NV	cube map target must be consistent
DOT_PRODUCT_REFLECT_CUBE_MAP_NV	cube map target must be consistent
DOT_PRODUCT_CONST_EYE_-REFLECT_CUBE_MAP_NV	cube map target must be consistent
DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV	cube map target must be consistent
DOT_PRODUCT_DEPTH_REPLACE_NV	-

**Table 3.B: Texture shader target dependencies for each operation.
If the dependency listed above is not met, the texture shader stage is
considered inconsistent.**

texture shader operation i	texture coordinate set usage	texture target	uses stage result i-1	uses stage result i-2	uses stage result i+1	uses previous texture input	uses cull modes	uses offset texture 2D matrix	offset texture 2D scale and bias	uses const eye vector
NONE	-	-	-	-	-	-	-	-	-	-
TEXTURE_1D	s,q	1D	-	-	-	-	-	-	-	-
TEXTURE_2D	s,t,q	2D	-	-	-	-	-	-	-	-
TEXTURE_RECTANGLE_NV	s,t,q	rectangle	-	-	-	-	-	-	-	-
TEXTURE_CUBE_MAP_ARB	s,t,r	cube map	-	-	-	-	-	-	-	-
PASS_THROUGH_NV	s,t,r,q	-	-	-	-	-	-	-	-	-
CULL_FRAGMENT_NV	s,t,r,q	-	-	-	-	-	y	-	-	-
OFFSET_TEXTURE_2D_NV	s,t	2D	-	-	-	y	-	y	-	-
OFFSET_TEXTURE_2D_SCALE_NV	s,t	2D	-	-	-	y	-	y	y	-
OFFSET_TEXTURE_RECTANGLE_NV	s,t	rectangle	-	-	-	y	-	y	-	-
OFFSET_TEXTURE_RECTANGLE_SCALE_NV	s,t	rectangle	-	-	-	y	-	y	y	-
DEPENDENT_AR_TEXTURE_2D_NV	-	2D	-	-	-	y	-	-	-	-
DEPENDENT_GB_TEXTURE_2D_NV	-	2D	-	-	-	y	-	-	-	-
DOT_PRODUCT_NV	s,t,r (q*)	-	-	-	-	y	-	-	-	-
DOT_PRODUCT_TEXTURE_2D_NV	s,t,r	2D	y	-	-	y	-	-	-	-
DOT_PRODUCT_TEXTURE_RECTANGLE_NV	s,t,r	rectangle	y	-	-	y	-	-	-	-
DOT_PRODUCT_TEXTURE_CUBE_MAP_NV	s,t,r	cube map	y	y	-	y	-	-	-	-
DOT_PRODUCT_REFLECT_CUBE_MAP_NV	s,t,r,q	cube map	y	y	-	y	-	-	-	-
DOT_PRODUCT_CONST_EYE_-REFLECT_CUBE_MAP_NV	s,t,r	cube map	y	y	-	y	-	-	-	y
DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV	s,t,r (q*)	cube map	y	y	y	y	-	-	-	-
DOT_PRODUCT_DEPTH_REPLACE_NV	s,t,r	-	y	-	-	y	-	-	-	-

Table 3.C: Relevant texture shader computation inputs for each operation. The (q*) for the texture coordinate set usage indicates that the q texture coordinate is used only when the DOT_PRODUCT_NV and DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV operations are used in conjunction with DOT_PRODUCT_REFLECT_CUBE_MAP_NV.

texture shader operation i	shader stage result type	shader stage result	texture unit RGBA color result
NONE	unsigned RGBA	invalid	(0,0,0,0)
TEXTURE_1D	matches 1D target type	filtered 1D target texel	if 1D target texture type is RGBA, filtered 1D target texel, else (0,0,0,0)
TEXTURE_2D	matches 2D target type	filtered 2D target texel	if 2D target texture type is RGBA, filtered 2D target texel, else (0,0,0,0)
TEXTURE_RECTANGLE_NV	matches rectangle target type	filtered rectangle target texel	if rectangle target texture type is RGBA, filtered rectangle target texel, else (0,0,0,0)
TEXTURE_CUBE_MAP_ARB	matches cube map target type	filtered cube map target texel	if cube map target texture type is RGBA, filtered cube map target texel, else (0,0,0,0)
PASS_THROUGH_NV	unsigned RGBA	(max(0,min(1,s)), max(0,min(1,t)), max(0,min(1,r)), max(0,min(1,q)))	(max(0,min(1,s)), max(0,min(1,t)), max(0,min(1,r)), max(0,min(1,q)))
CULL_FRAGMENT_NV	unsigned RGBA	invalid	(0,0,0,0)
OFFSET_TEXTURE_2D_NV	matches 2D target type	filtered 2D target texel	if 2D target texture type is RGBA, filtered 2D target texel, else (0,0,0,0)
OFFSET_TEXTURE_2D_SCALE_NV	unsigned RGBA	filtered 2D target texel	scaled filtered 2D target texel
OFFSET_TEXTURE_RECTANGLE_NV	matches rectangle target type	filtered rectangle target texel	if rectangle target texture type is RGBA, filtered rectangle target texel, else (0,0,0,0)
OFFSET_TEXTURE_RECTANGLE_SCALE_NV	unsigned RGBA	filtered rectangle target texel	scaled filtered rectangle target texel
DEPENDENT_AR_TEXTURE_2D_NV	matches 2D target type	filtered 2D target texel	if 2D target texture type is RGBA, filtered 2D target texel, else (0,0,0,0)
DEPENDENT_GB_TEXTURE_2D_NV	matches 2D target type	filtered 2D target texel	if 2D target texture type is RGBA, filtered 2D target texel, else (0,0,0,0)
DOT_PRODUCT_NV	float	dot product	(0,0,0,0)
DOT_PRODUCT_TEXTURE_2D_NV	matches 2D target type	filtered 2D target texel	if 2D target texture type is RGBA, filtered 2D target texel, else (0,0,0,0)
DOT_PRODUCT_TEXTURE_RECTANGLE_NV	matches rectangle target type	filtered rectangle target texel	if rectangle target texture type is RGBA, filtered rectangle target texel, else (0,0,0,0)
DOT_PRODUCT_TEXTURE_CUBE_MAP_NV	matches cube map target type	filtered cube map target texel	if cube map target texture type is RGBA, filtered cube map target texel, else (0,0,0,0)
DOT_PRODUCT_REFLECT_CUBE_MAP_NV	matches cube map target type	filtered cube map target texel	if cube map target texture type is RGBA, filtered cube map target texel, else (0,0,0,0)
DOT_PRODUCT_CONST_EYE_-REFLECT_CUBE_MAP_NV	matches cube map target type	filtered cube map target texel	if cube map target texture type is RGBA, filtered cube map target texel, else (0,0,0,0)
DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV	matches cube map target type	filtered cube map target texel	if cube map target texture type is RGBA, filtered cube map target texel, else (0,0,0,0)
DOT_PRODUCT_DEPTH_REPLACE_NV	unsigned RGBA	invalid	(0,0,0,0)

Table 3.D: Texture shader stage results for each operation.

Base internal format	Red	Green	Blue	Alpha
ALPHA	1	1	1	At
LUMINANCE	Lt	Lt	Lt	1
INTENSITY	It	It	It	It
LUMINANCE_ALPHA	Lt	Lt	Lt	At
RGB	Rt	Gt	Bt	1
RGBA	Rt	Gt	Bt	At

Table 3.E: How base internal formats components are mapped to RGBA values for texture shaders (note that the mapping for ALPHA is different from the mapping in Table 3.23 in the EXT_texture_env_combine extension).

3.8.13.1.1 None

The NONE texture shader operation ignores the texture unit's texture coordinate set and always generates the texture unit RGBA result (0,0,0,0) for its filtered texel value. The texture shader result is invalid. This texture shader stage is always consistent.

When a texture unit is not needed while texture shaders are enabled, it is most efficient to set the texture unit's texture shader operation to NONE.

3.8.13.1.2 1D Projective Texturing

The TEXTURE_1D texture shader operation accesses the texture unit's 1D texture object (as described in sections 3.8.4, 3.8.5, and 3.8.6) using (s/q) for the 1D texture coordinate where s and q are the homogeneous texture coordinates for the texture unit. The result of the texture access becomes both the shader result and texture unit RGBA result (see table 3.E). The type of the shader result depends on the format type of the accessed texture. This mode is equivalent to conventional texturing's 1D texture target.

If the texture unit's 1D texture object is not consistent, then this texture shader stage is not consistent.

If this texture shader stage is not consistent, it operates as if it is the NONE operation.

3.8.13.1.3 2D Projective Texturing

The TEXTURE_2D texture shader operation accesses the texture unit's 2D texture object (as described in sections 3.8.4, 3.8.5, and 3.8.6) using (s/q,t/q) for the 2D texture coordinates where s, t, and q are the homogeneous texture coordinates for the texture unit. The result of the texture access becomes both the shader result and texture unit RGBA result (see table 3.E). The type of the shader result depends on the format type of the accessed texture. This mode is equivalent to conventional texturing's 2D texture target.

If the texture unit's 2D texture object is not consistent, then this texture shader stage is not consistent.

If this texture shader stage is not consistent, it operates as if it is the NONE operation.

3.8.13.1.4 Rectangle Projective Texturing

The TEXTURE_RECTANGLE_NV texture shader operation accesses the texture unit's rectangle texture object (as described in sections 3.8.4, 3.8.5, and 3.8.6) using (s/q,t/q) for the 2D texture coordinates where s, t, and q are the homogeneous texture coordinates for the texture unit. The result of the texture access becomes both the shader result and texture unit RGBA result (see table 3.E). The type of the shader result depends on the format type of the accessed texture. This mode is equivalent to NV_texture_rectangle's rectangle texture target.

If the texture unit's rectangle texture object is not consistent, then this texture shader stage is not consistent.

If this texture shader stage is not consistent, it operates as if it is the NONE operation.

3.8.13.1.5 Cube Map Texturing

The TEXTURE_CUBE_MAP_ARB texture shader operation accesses the texture unit's cube map texture object (as described in the ARB_texture_cube_map specification) using (s,t,r) for the 3D texture coordinate where s, t, and r are the homogeneous texture coordinates for the texture unit. The result of the texture access becomes both the shader result and texture unit RGBA result (see table 3.E). The type of the shader result depends on the format type of the accessed texture. This mode is equivalent to conventional texturing's cube map texture target.

If the texture unit's cube map texture object is not consistent, then this texture shader stage is not consistent.

If this texture shader stage is not consistent, it operates as if it is the NONE operation.

3.8.13.1.6 Pass Through

The PASS_THROUGH_NV texture shader operation converts an (s,t,r,q) texture coordinate set into an RGBA color result (r,g,b,a). Each texture coordinate is first clamped to [0,1] before being mapped to its corresponding color component. The texture shader result and texture unit RGBA result of this operation are both assigned the clamped RGBA color result.

This operation in no way depends on any of the texture unit's texture objects.

3.8.13.1.7 Cull Fragment

The CULL_FRAGMENT_NV texture shader operation compares each component of the texture coordinate set (s,t,r,q) to zero based on the texture shader's corresponding cull mode. For the LESS cull mode to succeed, the corresponding component must be less than zero; otherwise the comparison fails. For the GEQUAL cull mode to succeed, the corresponding component must be greater or equal to zero; otherwise the comparison fails. If any of the four comparisons fails, the fragment is discarded.

The texture unit RGBA result generated is always (0,0,0,0). The texture shader result is invalid. This texture shader stage is always consistent.

This operation in no way depends on any of the texture unit's texture objects.

3.8.13.1.8 Offset Texture 2D

The OFFSET_TEXTURE_2D_NV texture shader operation uses the transformed result of a previous texture shader stage to perturb the current texture shader stage's (s,t) texture coordinates (without a projective division by q). The resulting perturbed texture coordinates (s',t') are used to access the texture unit's 2D texture object (as described in sections 3.8.4, 3.8.5, and 3.8.6).

The result of the texture access becomes both the shader result and texture unit RGBA result (see table 3.E). The type of the shader result depends on the format type of the accessed texture.

The perturbed texture coordinates s' and t' are computed with floating-point math as follows:

```
  s' = s + a1 * DSprev + a3 * DTprev
  t' = t + a2 * DSprev + a4 * DTprev
```

where a1, a2, a3, and a4 are the texture shader stage's OFFSET_TEXTURE_MATRIX_NV values, and DSprev and DTprev are the (signed) DS and DT components of a previous texture shader unit's texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value.

If the texture unit's 2D texture object is not consistent, then this texture shader stage is not consistent.

If the previous texture input texture object specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value has a base internalformat that is not one of DSDT_NV, DSDT_MAG_NV or DSDT_MAG_INTENSITY_NV, then this texture shader stage is not consistent.

If the previous texture input texture shader operation specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is DOT_PRODUCT_NV, then this texture shader stage is not consistent.

If the previous texture input texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is invalid, then this texture shader stage is not consistent.

If the previous texture input shader stage specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is not consistent, then this texture shader stage is not consistent.

If this texture shader stage is not consistent, it operates as if it is the NONE operation.

3.8.13.1.9 Offset Texture 2D and Scale

The OFFSET_TEXTURE_2D_SCALE_NV texture shader operation extends the functionality of the OFFSET_TEXTURE_2D_NV texture shader operation. The texture unit's 2D texture object is accessed by the same perturbed s' and t' coordinates used by the OFFSET_TEXTURE_2D_NV operation. The red, green, and blue components (but not alpha) of the RGBA result of the texture access are further scaled by the value Scale and clamped to the range [0,1]. This RGBA result is this shader's texture unit RGBA result. This shader's texture shader result is the RGBA result of the texture access prior to scaling and clamping.

Scale is computed with floating-point math as follows:

```
  Scale = max(0.0, min(1.0, textureOffsetBias + textureOffsetScale * MAGprev))
```

where textureOffsetBias is the texture shader stage's OFFSET_TEXTURE_BIAS_NV value, textureOffsetScale is the texture shader stage's OFFSET_TEXTURE_SCALE_NV value, and MAGprev is the magnitude component of the a previous texture shader unit's result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value.

The texture unit RGBA result (red',green',blue',alpha') is computed as follows:

```
  red'   = Scale * red
  green' = Scale * green
  blue'  = Scale * blue
  alpha' = alpha
```

where red, green, blue, and alpha are the texture access components.

If the unit's 2D texture object has any signed components, then this texture shader stage is not consistent.

If the texture unit's 2D texture object is has a format type other than RGBA (the DSDT_MAG_INTENSITY_NV base internal format does not count as an RGBA format type in this context), then this texture shader stage is not consistent.

If the texture unit's 2D texture object is not consistent, then this texture shader stage is not consistent.

If the previous texture input texture object specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value has a base internalformat that is not either DSDT_MAG_NV or DSDT_MAG_INTENSITY_NV, then this texture shader stage is not consistent.

If the previous texture input texture shader operation specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is DOT_PRODUCT_NV, then this texture shader stage is not consistent.

If the previous texture input texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is invalid, then this texture shader stage is not consistent.

If the previous texture input shader stage specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is not consistent, then this texture shader stage is not consistent.

If this texture shader stage is not consistent, it operates as if it is the NONE operation.

3.8.13.1.10 Offset Texture Rectangle

The OFFSET_TEXTURE_RECTANGLE_NV shader operation operates identically to the OFFSET_TEXTURE_2D_NV shader operation except that the rectangle texture target is accessed rather than the 2D texture target.

If the texture unit's rectangle texture object (rather than the 2D texture object) is not consistent, then this texture shader stage is not consistent.

3.8.13.1.11 Offset Texture Rectangle Scale

The OFFSET_TEXTURE_RECTANGLE_SCALE_NV shader operation operates identically to the OFFSET_TEXTURE_2D_SCALE_NV shader operation except that the rectangle texture target is accessed rather than the 2D texture target.

If the texture unit's rectangle texture object (rather than the 2D texture object) is not consistent, then this texture shader stage is not consistent.

3.8.13.1.12 Dependent Alpha-Red Texturing

The DEPENDENT_AR_TEXTURE_2D_NV texture shader operation accesses the texture unit's 2D texture object (as described in section 3.8.4, 3.8.5, and 3.8.6) using (Aprev, Rprev) for the 2D texture coordinates where Aprev and Rprev are the are the alpha and red components of a previous texture input's RGBA texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value. The result of the texture access becomes both the shader result and texture unit RGBA result (see table 3.E). The type of the shader result depends on the format type of the accessed texture.

If the texture unit's 2D texture object is not consistent, then this texture shader stage is not consistent.

If the previous texture input's texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value has a texture shader result type other than RGBA (the DSDT_MAG_INTENSITY_NV base internal format does not count as an RGBA format type in this context), then this texture shader stage is not consistent.

If the previous texture input's texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value has a texture shader result type of RGBA but any of the RGBA components are signed, then this texture shader stage is not consistent.

If the previous texture input texture shader operation specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is DOT_PRODUCT_NV, then this texture shader stage is not consistent.

If the previous texture input texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is invalid, then this texture shader stage is not consistent.

If the previous texture input shader stage specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is not consistent, then this texture shader stage is not consistent.

If this texture shader stage is not consistent, it operates as if it is the NONE operation.

3.8.13.1.13 Dependent Green-Blue Texturing

The DEPENDENT_GB_TEXTURE_2D_NV texture shader operation accesses the texture unit's 2D texture object (as described in section 3.8.4, 3.8.5, and 3.8.6) using (Gprev, Bprev) for the 2D texture coordinates where Gprev and Bprev are the are the green and blue components of a previous texture input's RGBA texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value. The result of the texture access becomes both the shader result and texture unit RGBA result (see table 3.E). The type of the shader result depends on the format type of the accessed texture.

If the texture unit's 2D texture object is not consistent, then this texture shader stage is not consistent.

If the previous texture input's texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value has a texture shader result type other than RGBA (the DSDT_MAG_INTENSITY_NV base internal format does not count as an RGBA format type in this context), then this texture shader stage is not consistent.

If the previous texture input's texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value has a texture shader result type of RGBA but any of the RGBA components are signed, then this texture shader stage is not consistent.

If the previous texture input texture shader operation specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is DOT_PRODUCT_NV, then this texture shader stage is not consistent.

If the previous texture input texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is invalid, then this texture shader stage is not consistent.

If the previous texture input shader stage specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is not consistent, then this texture shader stage is not consistent.

If this texture shader stage is not consistent, it operates as if it is the NONE operation.

3.8.13.1.14 Dot Product

The DOT_PRODUCT_NV texture shader operation computes a floating-point texture shader result. The texture shader result is the floating-point dot product of the texture unit's (s,t,r) texture coordinates and a remapped version of the RGBA or HILO texture shader result from a specified previous texture shader stage. The RGBA color result of this shader is always (0,0,0,0).

The re-mapping depends on the specified previous texture shader stage's texture shader result type. Specifically, the re-mapping depends on whether this texture shader result type has all signed components or all unsigned components, and whether it has RGBA components or HILO components, and, in the case of unsigned RGBA texture shader results, the RGBA_UNSIGNED_DOT_PRODUCT_MAPPING_NV state.

If the specified previous texture unit's texture shader result type is HILO and all the type components are unsigned, then the floating-point result is computed by

```
  result = s * HI + t * LO + r
```

where HI and LO are the (unsigned) hi and lo components respectively of the previous texture unit's HILO texture shader result.

If the specified previous texture unit's texture shader result type is HILO and all the type components are signed, then the floating-point result is computed by

```
  result = s * HI + t * LO + r * sqrt(max(0, 1.0 - HI*HI - LO*LO))
```

where HI and LO are the (signed) hi and lo components respectively of the previous texture unit's texture shader result.

If the specified previous texture unit's texture shader result contains only signed RGBA components, then the floating-point result is computed by

```
  result = s * Rprev + t * Gprev + r * Bprev
```

where Rprev, Gprev, and Bprev are the (signed) red, green, and blue components respectively of the previous texture unit's RGBA texture shader result.

If the specified previous texture unit's texture shader result contains only unsigned RGBA components, then the dot product computation depends on the RGBA_UNSIGNED_DOT_PRODUCT_MAPPING_NV state. When the RGBA_UNSIGNED_DOT_PRODUCT_MAPPING_NV is UNSIGNED_IDENTITY_NV, then the floating-point result for unsigned RGBA components is computed by

```
  result = s * Rprev + t * Gprev + r * Bprev
```

where Rprev, Gprev, and Bprev are the (unsigned) red, green, and blue components respectively of the previous texture unit's RGBA texture shader result.

When the RGBA_UNSIGNED_DOT_PRODUCT_MAPPING_NV is EXPAND_NORMAL_NV, then the floating-point result for unsigned RGBA components is computed by

```
  result = s * (2.0*Rprev-1.0) + t * (2.0*Gprev-1.0) + r * (2.0*Bprev-1.0)
```

where Rprev, Gprev, and Bprev are the (unsigned) red, green, and blue components respectively of the previous texture unit's RGBA texture shader result.

If the previous texture input texture object specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value has a format type other than RGBA or HILO (the DSDT_MAG_INTENSITY_NV base internal format does not count as an RGBA format type in this context), then this texture shader stage is not consistent.

If the components of the previous texture input texture object specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value have mixed signedness, then this texture shader stage is not consistent. For example, the SIGNED_RGB_UNSIGNED_ALPHA_NV base internal format has mixed signedness.

If the previous texture input texture shader operation specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is DOT_PRODUCT_NV, then this texture shader stage is not consistent.

If the previous texture input texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is invalid, then this texture shader stage is not consistent.

If the previous texture input shader stage specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is not consistent, then this texture shader stage is not consistent.

If this texture shader stage is not consistent, it operates as if it is the NONE operation.

This operation in no way depends on any of the texture unit's texture objects.

3.8.13.1.15 Dot Product Texture 2D

The DOT_PRODUCT_TEXTURE_2D_NV texture shader operation accesses the texture unit's 2D texture object (as described in sections 3.8.4, 3.8.5, and 3.8.6) using (dotP,dotC) for the 2D texture coordinates. The result of the texture access becomes both the shader result and texture unit RGBA result (see table 3.E). The type of the shader result depends on the format type of the accessed texture.

Assuming that i is the current texture shader stage, dotP is the floating-point dot product result from the i-1 texture shader stage, assuming the i-1 texture shader stage's operation is DOT_PRODUCT_NV. dotC is the floating-point dot product result from the current texture shader stage. dotC is computed in the identical manner used to compute the floating-point result of the DOT_PRODUCT_NV texture shader described in section 3.8.13.1.14.

If the previous texture input texture object specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value has a format type other than RGBA or HILO (the DSDT_MAG_INTENSITY_NV base internal format does not count as an RGBA format type in this context), then this texture shader stage is not consistent.

If the previous texture input texture shader operation specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is DOT_PRODUCT_NV, then this texture shader stage is not consistent.

If the previous texture input texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is invalid, then this texture shader stage is not consistent.

If the previous texture input shader stage specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is not consistent, then this texture shader stage is not consistent.

If the i-1 texture shader stage operation is not DOT_PRODUCT_NV, then this texture shader stage is not consistent.

If the i-1 texture shader stage is not consistent, then this texture shader stage is not consistent.

If the texture unit's 2D texture object is not consistent, then this texture shader stage is not consistent.

If this texture shader stage is not consistent, it operates as if it is the NONE operation.

3.8.13.1.16 Dot Product Texture Rectangle

The DOT_PRODUCT_TEXTURE_RECTANGLE_NV shader operation operates identically to the DOT_PRODUCT_TEXTURE_2D_NV shader operation except that the rectangle texture target is accessed rather than the 2D texture target.

If the texture unit's rectangle texture object (rather than the 2D texture object) is not consistent, then this texture shader stage is not consistent.

3.8.13.1.17 Dot Product Texture Cube Map

The DOT_PRODUCT_TEXTURE_CUBE_MAP_NV texture shader operation accesses the texture unit's cube map texture object (as described in the ARB_texture_cube_map specification) using (dotPP,dotP,dotC) for the 3D texture coordinates. The result of the texture access becomes both the shader result and texture unit RGBA result (see table 3.E). The type of the shader result depends on the format type of the accessed texture.

Assuming that i is the current texture shader stage, dotPP is the floating-point dot product texture shader result from the i-2 texture shader stage, assuming the i-2 texture shader stage's operation is DOT_PRODUCT_NV. dotP is the floating-point dot product texture shader result from the i-1 texture shader stage, assuming the i-1 texture shader stage's operation is DOT_PRODUCT_NV. dotC is the floating-point dot product result from the current texture shader stage. dotC is computed in the identical manner used to compute the floating-point result of the DOT_PRODUCT_NV texture shader described in section 3.8.13.1.14.

If the previous texture input texture object specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value has a format type other than RGBA or HILO (the DSDT_MAG_INTENSITY_NV base internal format does not count as an RGBA format type in this context), then this texture shader stage is not consistent.

If the previous texture input texture shader operation specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is DOT_PRODUCT_NV, then this texture shader stage is not consistent.

If the previous texture input texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is invalid, then this texture shader stage is not consistent.

If the previous texture input shader stage specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is not consistent, then this texture shader stage is not consistent.

If either the i-1 or i-2 texture shader stage operation is not DOT_PRODUCT_NV, then this texture shader stage is not consistent.

If either the i-1 or i-2 texture shader stage is not consistent, then this texture shader stage is not consistent.

If the texture unit's cube map texture object is not consistent, then this texture shader stage is not consistent.

If this texture shader stage is not consistent, it operates as if it is the NONE operation.

3.8.13.1.18 Dot Product Reflect Cube Map

The DOT_PRODUCT_REFLECT_CUBE_MAP_NV and DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV (described in the section 3.8.13.1.20) texture shader operations are typically used together.

The DOT_PRODUCT_REFLECT_CUBE_MAP_NV texture shader operation accesses the texture unit's cube map texture object (as described in the ARB_texture_cube_map specification) using (rx,ry,rz) for the 3D texture coordinates. The result of the texture access becomes both the shader result and texture unit RGBA result (see table 3.E). The type of the shader result depends on the format type of the accessed texture.

Let R = (rx,ry,rz), N = (dotPP,dotP,dotC), and E = (qPP,qP,qC), then

```
  R = 2 * (N dot E) / (N dot N) * N - E
```

Assuming that i is the current texture shader stage, dotPP is the floating-point dot product texture shader result from the i-2 texture shader stage, assuming the i-2 texture shader stage's operation is DOT_PRODUCT_NV. dotP is the floating-point dot product texture shader result from the i-1 texture shader stage, assuming the i-1 texture shader stage's operation is either DOT_PRODUCT_NV or DOT_PRODUCT_DIFFUSE_NV. dotC is the floating-point dot product result from the current texture shader stage. dotC is computed in the identical manner used to compute the floating-point result of the DOT_PRODUCT_NV texture shader described in section 3.8.13.1.14.

qPP is the q component of the i-2 texture shader stage's texture coordinate set. qP is the q component of the i-1 texture shader stage's texture coordinate set. qC is the q component of the current texture shader stage's texture coordinate set.

If the previous texture input texture object specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value has a format type other than RGBA or HILO (the DSDT_MAG_INTENSITY_NV base internal format does not count as an RGBA format type in this context), then this texture shader stage is not consistent.

If the previous texture input texture shader operation specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is DOT_PRODUCT_NV, then this texture shader stage is not consistent.

If the previous texture input texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is invalid, then this texture shader stage is not consistent.

If this texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value refers to texture unit i-2 or i-1, then this texture shader stage is not consistent.

If the previous texture input shader stage specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is not consistent, then this texture shader stage is not consistent.

If the i-2 texture shader stage operation is not DOT_PRODUCT_NV, then this texture shader stage is not consistent.

If the i-1 texture shader stage operation is not DOT_PRODUCT_NV or DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV, then this texture shader stage is not consistent.

If either the i-1 or i-2 texture shader stage is not consistent, then this texture shader stage is not consistent.

If the texture unit's cube map texture object is not consistent, then this texture shader stage is not consistent.

If this texture shader stage is not consistent, it operates as if it is the NONE operation.

3.8.13.1.19 Dot Product Constant Eye Reflect Cube Map

The DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV texture shader operation operates the same as the DOT_PRODUCT_REFLECT_CUBE_MAP_NV operation except that the eye vector E is equal to the three floating-point values assigned to the texture shader's eye constant (rather than the three q components of the given texture unit and the previous two texture units).

The DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV operation has the same texture shader consistency rules as the DOT_PRODUCT_REFLECT_CUBE_MAP_NV operation.

3.8.13.1.20 Dot Product Diffuse Cube Map

The DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV texture shader operation accesses the texture unit's cube map texture object (as described in the ARB_texture_cube_map specification) using (dotP,dotC,dotN) for the 3D texture coordinates. The result of the texture access becomes both the shader result and texture unit RGBA result (see table 3.E). The type of the shader result depends on the format type of the accessed texture.

Assuming that i is the current texture shader stage, dotP is the floating-point dot product texture shader result from the i-1 texture shader stage, assuming the i-1 texture shader stage's operation is DOT_PRODUCT_NV. dotC is the floating-point dot product result from the current texture shader stage. dotC is computed in the identical manner used to compute the floating-point result of the DOT_PRODUCT_NV texture shader described in section 3.8.13.1.14. dotN is the floating-point dot product texture shader result from the i+1 texture shader stage, assuming the next texture shader stage's operation is either DOT_PRODUCT_REFLECT_CUBE_MAP_NV or DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV.

If the texture unit's cube map texture object is not consistent, then this operation operates as if it is the NONE operation. If the previous texture unit's texture shader operation is not DOT_PRODUCT_NV, then this operation operates as if it is the NONE operation. If the next texture unit's texture shader operation is neither DOT_PRODUCT_REFLECT_CUBE_MAP_NV nor DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV, then this operation operates as if it is the NONE operation. If the next texture unit's texture shader operation is either DOT_PRODUCT_REFLECT_CUBE_MAP_NV or DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV, but the next texture unit operation is operating as if it is the NONE operation, then this operation operates as if it is the NONE operation. If the specified previous input texture unit is inconsistent or uses the DOT_PRODUCT_NV texture shader operation, then this operation operates as if it is the NONE operation.

If the previous texture input texture object specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value has a format type other than RGBA or HILO (the DSDT_MAG_INTENSITY_NV base internal format does not count as an RGBA format type in this context), then this texture shader stage is not consistent.

If the previous texture input texture shader operation specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is DOT_PRODUCT_NV, then this texture shader stage is not consistent.

If the previous texture input texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is invalid, then this texture shader stage is not consistent.

If the previous texture input shader stage specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is not consistent, then this texture shader stage is not consistent.

If the i-1 texture shader stage operation is not DOT_PRODUCT_NV, then this texture shader stage is not consistent.

If the i+1 texture shader stage operation is not DOT_PRODUCT_REFLECT_CUBE_MAP_NV or DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV, then this texture shader stage is not consistent.

If either the i-1 or i+1 texture shader stage is not consistent, then this texture shader stage is not consistent.

If the texture unit's cube map texture object is not consistent, then this texture shader stage is not consistent.

If this texture shader stage is not consistent, it operates as if it is the NONE operation.

3.8.13.1.21 Dot Product Depth Replace

The DOT_PRODUCT_DEPTH_REPLACE_NV texture shader operation replaces the incoming fragments depth (in window coordinates, after polygon offset and before conversion to fixed-point, i.e. in the [0,1] range) with a new depth value. The new depth is computed as follows:

```
  depth = dotP / dotC
```

Assuming that i is the current texture shader stage, dotP is the floating-point dot product texture shader result from the i-1 texture shader stage, assuming the i-1 texture shader stage's operation is DOT_PRODUCT_NV. dotC is the floating-point dot product result from the current texture shader stage. dotC is computed in the identical manner used to compute the floating-point result of the DOT_PRODUCT_NV texture shader described in section 3.8.13.1.14.

If the new depth value is outside of the range of the near and far depth range values, the fragment is rejected.

The texture unit RGBA result generated is always (0,0,0,0). The texture shader result is invalid.

If the previous texture input texture object specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value has a format type other than RGBA or HILO (the DSDT_MAG_INTENSITY_NV base internal format does not count as an RGBA format type in this context), then this texture shader stage is not consistent.

If the previous texture input texture shader operation specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is DOT_PRODUCT_NV, then this texture shader stage is not consistent.

If the previous texture input texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is invalid, then this texture shader stage is not consistent.

If the previous texture input shader stage specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is not consistent, then this texture shader stage is not consistent.

If the i-1 texture shader stage operation is not DOT_PRODUCT_NV, then this texture shader stage is not consistent.

If the i-1 texture shader stage is not consistent, then this texture shader stage is not consistent.

If any previous texture shader stage operation is DOT_PRODUCT_DEPTH_REPLACE_NV and that previous stage is consistent, then this texture shader stage is not consistent. (This eliminates the potential for two stages to each be performing a depth replace operation.)

If this texture shader stage is not consistent, it operates as if it is the NONE operation.

This operation in no way depends on any of the texture unit's texture objects.

3.8.13.2 Texture Shader Restrictions

There are various restrictions on possible texture shader configurations. These restrictions are described in this section.

The error INVALID_OPERATION occurs if the SHADER_OPERATION_NV parameter for texture unit 0 is assigned one of OFFSET_TEXTURE_2D_NV, OFFSET_TEXTURE_2D_SCALE_NV, OFFSET_TEXTURE_RECTANGLE_NV, OFFSET_TEXTURE_RECTANGLE_SCALE_NV, DEPENDENT_AR_TEXTURE_2D_NV, DEPENDENT_GB_TEXTURE_2D_NV, DOT_PRODUCT_NV, DOT_PRODUCT_DEPTH_REPLACE_NV, DOT_PRODUCT_TEXTURE_2D_NV, DOT_PRODUCT_TEXTURE_RECTANGLE_NV, DOT_PRODUCT_TEXTURE_CUBE_MAP_NV, DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV, DOT_PRODUCT_REFLECT_CUBE_MAP_NV, or DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV. Each of these texture shaders requires a previous texture shader result that is not possible for texture unit 0. Therefore these shaders are disallowed for texture unit 0.

The error INVALID_OPERATION occurs if the SHADER_OPERATION_NV parameter for texture unit 1 is assigned one of DOT_PRODUCT_DEPTH_REPLACE_NV, DOT_PRODUCT_TEXTURE_2D_NV, DOT_PRODUCT_TEXTURE_RECTANGLE_NV, DOT_PRODUCT_TEXTURE_CUBE_MAP_NV, DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV, DOT_PRODUCT_REFLECT_CUBE_MAP_NV, or DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV. Each of these texture shaders requires either two previous texture shader results or a dot product result that cannot be generated by texture unit 0. Therefore these shaders are disallowed for texture unit 1.

The error INVALID_OPERATION occurs if the SHADER_OPERATION_NV parameter for texture unit 2 is assigned one of

DOT_PRODUCT_TEXTURE_CUBE_MAP_NV, DOT_PRODUCT_REFLECT_CUBE_MAP_NV, DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV. Each of these texture shaders requires three previous texture shader results. Therefore these shaders are disallowed for texture unit 2.

The error INVALID_OPERATION occurs if the SHADER_OPERATION_NV parameter for texture unit n-1 (where n is the number of supported texture units) is assigned either DOT_PRODUCT_NV or DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV. DOT_PRODUCT_NV is invalid for the final texture shader stage because it is only useful as an input to a successive texture shader stage. DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV is invalid for the final texture shader stage because it must be followed by the DOT_PRODUCT_REFLECT_CUBE_MAP_NV operation in the immediately successive stage. Therefore these shaders are disallowed for texture unit n-1.

3.8.13.3 Required State

The state required for texture shaders consists of a single bit to indicate whether or not texture shaders are enabled, a vector of three floating-point values for the constant eye vector, and n sets of per-texture unit state where n is the implementation-dependent number of supported texture units. The set of per-texture unit texture shader state consists of the twenty-one-valued integer indicating the texture shader operation, four two-valued integers indicating the cull modes, an integer indicating the previous texture unit input, a two-valued integer indicating the RGBA unsigned dot product mapping mode, a 2x2 floating-point matrix indicating the texture offset transform, a floating-point value indicating the texture offset scale, a floating-point value indicating the texture offset bias, and a bit to indicate whether or not the texture shader stage is consistent.

In the initial state, the texture shaders state is set as follows: the texture shaders enable is disabled; the constant eye vector is (0,0,-1); all the texture shader operations are NONE; the RGBA unsigned dot product mapping mode is UNSIGNED_IDENTITY_NV; all the cull mode values are GEQUAL; all the previous texture units are TEXTURE0_ARB; each texture offset matrix is an identity matrix; all texture offset scales are 1.0; and all texture offset biases are 0.0."

Additions to Chapter 4 of the 1.2 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.2 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

-- Section 6.1.3 "Texture Environments and Texture Functions"

Change the third sentence of the third paragraph to read:

"The env argument to GetTexEnv must be one of TEXTURE_ENV, TEXTURE_FILTER_CONTROL_EXT, or TEXTURE_SHADER_NV."

Add to the end of the third paragraph:

"For GetTexEnv, when the target is TEXTURE_SHADER_NV, the texture shader stage consistency can be queried with SHADER_CONSISTENT_NV."

Change the following sentence in the fouth paragraph to include sizes for the newly introduced component:

"Queries of TEXTURE_RED_SIZE, TEXTURE_GREEN_SIZE, TEXTURE_BLUE_SIZE, TEXTURE_ALPHA_SIZE, TEXTURE_LUMINANCE_SIZE, TEXTURE_DS_SIZE_EXT, TEXTURE_DT_SIZE_EXT, TEXTURE_HI_SIZE_EXT, TEXTURE_LO_SIZE_EXT, TEXTURE_MAG_SIZE_EXT, and TEXTURE_INTENSITY_SIZE return the actual resolutions of the stored image array components, not the resolutions specified when the image array was defined."

Add the following to the end of the fourth paragraph:

"Queries of TEXTURE_BORDER_COLOR return the same values as the TEXTURE_BORDER_VALUES query."

-- Section 6.1.4 "Texture Queries"

Add the following to the end of the fourth paragraph:

"Calling GetTexImage with a color format (one of RED, GREEN, BLUE, ALPHA, RGB, RGBA, BGR, BGRA, LUMINANCE, or LUMINANCE_ALPHA) when the texture image is of a format type (see table 3.15) other than RGBA (the DSDT_MAG_INTENSITY_NV base internal format does not count as an RGBA format type in this context) causes the error INVALID_OPERATION. Calling GetTexImage with a format of HILO_NV when the texture image is of a format type (see table 3.15) other than HILO_NV causes the error INVALID_OPERATION. Calling GetTexImage with a format of DSDT_NV when the texture image is of a base internal format other than DSDT_NV causes the error INVALID_OPERATION. Calling GetTexImage with a format of DSDT_MAG_NV when the texture image is of a base internal format other than DSDT_MAG_NV causes the error INVALID_OPERATION. Calling GetTexImage with a format of DSDT_MAG_VIB_NV when the texture image is of a base internal format other than DSDT_MAG_INTENSITY_NV causes the error INVALID_OPERATION."

Additions to the GLX Specification

None

Dependencies on ARB_texture_env_add or EXT_texture_env_add

If neither ARB_texture_env_add nor EXT_texture_env_add are implemented, then the references to ADD are invalid and should be ignored.

Dependencies on ARB_texture_env_combine or EXT_texture_env_combine

If neither ARB_texture_env_combine nor EXT_texture_env_combine are implemented, then the references to COMBINE_ARB and COMBINE_EXT are invalid and should be ignored.

Dependencies on EXT_texture_lod_bias

If EXT_texture_lod_bias is not implemented, then the references to TEXTURE_FILTER_CONTROL_EXT are invalid and should be ignored.

Dependencies on NV_texture_env_combine4

If NV_texture_env_combine4 is not implemented, then the references to COMBINE4_NV are invalid and should be ignored.

Dependencies on NV_texture_rectangle

If NV_texture_rectangle is not implemented, then the references to TEXTURE_RECTANGLE_NV, OFFSET_TEXTURE_RECTANGLE_NV, OFFSET_TEXTURE_RECTANGLE_SCALE_NV, and DOT_PRODUCT_TEXTURE_RECTANGLE_NV are invalid and should be ignored.

Dependencies on ARB_color_buffer_float

If ARB_color_buffer_float is also implemented, then the "max(0,x)", "max(-1,x)" and "min(1,x)" functions for clamping in tables 3.18 and 3.19 simply return "x" without applying the maximum or minimum function when CLAMP_FRAGMENT_COLOR_ARB is either FIXED_ONLY_ARB when rendering to a floating-point color framebuffer or FALSE.

However clamping operations for texture shader operations (specifically PASS_THROUGH_NV and OFFSET_TEXTURE_2D_SCALE_NV) are performed independent of the CLAMP_FRAGMENT_COLOR_ARB state.

The intent of these interactions is to eliminate the specified clamping behavior of texture environment functions when CLAMP_FRAGMENT_COLOR_ARB indicates clamping should not be performed.

Errors

INVALID_ENUM is generated if one of HILO_NV, DSDT_NV, DSDT_MAG_NV, or DSDT_MAG_VIBRANCE_NV is used as the format for DrawPixels, ReadPixels, ColorTable, ColorSubTable, ConvolutionFilter1D, ConvolutionFilter2D, SeparableFilter2D, GetColorTable, GetConvolutionFilter, GetSeparableFilter, GetHistogram, or GetMinmax.

INVALID_ENUM is generated if either UNSIGNED_INT_S8_S8_8_8_NV or UNSIGNED_INT_8_8_S8_S8_REV is used as the type for DrawPixels, ReadPixels, ColorTable, ColorSubTable, ConvolutionFilter1D, ConvolutionFilter2D, SeparableFilter2D, GetColorTable, GetConvolutionFilter, GetSeparableFilter, GetHistogram, or GetMinmax.

INVALID_OPERATION is generated if a packed pixel format type listed in table 3.8 is used with DrawPixels, ReadPixels, ColorTable, ColorSubTable, ConvolutionFilter1D, ConvolutionFilter2D, SeparableFilter2D, GetColorTable, GetConvolutionFilter, GetSeparableFilter, GetHistogram, GetMinmax, TexImage1D, TexImage2D, TexSubImage1D, TexSubImage2D, TexSubImage3d, or GetTexImage but the format parameter does not match on of the allowed Matching Pixel Formats listed in table 3.8 for the specified packed type parameter.

INVALID_OPERATION is generated when TexImage1D or TexImage2D are called and the format is HILO_NV and the internalformat is not one of HILO_NV, HILO16_NV, SIGNED_HILO_NV, SIGNED_HILO16_NV; or if the internalformat is one of HILO_NV, HILO16_NV, SIGNED_HILO_NV, or SIGNED_HILO16_NV and the format is not HILO_NV.

INVALID_OPERATION is generated when TexImage2D, or TexImage1D is called and if the format is DSDT_NV and the internalformat is not either DSDT_NV or DSDT8_NV; or if the internal format is either DSDT_NV or DSDT8_NV and the format is not DSDT_NV.

INVALID_OPERATION is generated when TexImage2D, or TexImage1D is called and if the format is DSDT_MAG_NV and the internalformat is not either DSDT_MAG_NV or DSDT8_MAG8_NV; or if the internal format is either DSDT_MAG_NV or DSDT8_MAG8_NV and the format is not DSDT_MAG_NV.

INVALID_OPERATION is generated when TexImage2D or TexImage1D is called and if the format is DSDT_MAG_VIB_NV and the internalformat is not either DSDT_MAG_INTENSITY_NV or DSDT8_MAG8_INTENSITY8_NV; or if the internal format is either DSDT_MAG_INTENSITY_NV or DSDT8_MAG8_INTENSITY8_NV and the format is not DSDT_MAG_VIB_NV.

INVALID_OPERATION is generated when CopyTexImage2D, CopyTexImage1D, CopyTexSubImage2D, or CopyTexSubImage1D is called and the internal format of the texture array to which the pixels are to be copied is one of HILO_NV, HILO16_NV, SIGNED_HILO_NV, SIGNED_HILO16_NV, DSDT_NV, DSDT8_NV, DSDT_MAG_NV, DSDT8_MAG8_NV, DSDT_MAG_INTENSITY_NV, or DSDT8_MAG8_INTENSITY8_NV.

INVALID_OPERATION is generated when TexSubImage2D or TexSubImage1D is called and the texture array's base internal format is not one of HILO_NV, DSDT_NV, DSDT_MAG_NV, or DSDT_INTENSITY_NV, and the format parameter is not one of COLOR_INDEX, RED, GREEN, BLUE, ALPHA, RGB, RGBA, LUMINANCE, or LUMINANCE_ALPHA

INVALID_OPERATION is generated when TexSubImage2D or TexSubImage1D is called and the texture array's base internal format is HILO_NV and the format parameter is not HILO_NV.

```
    INVALID_OPERATION is generated when TexSubImage2D or
    TexSubImage1D is called and the texture array's base internal format
    is DSDT_NV and the format parameter is not DSDT_NV.

    INVALID_OPERATION is generated when TexSubImage2D or
    TexSubImage1D is called and the texture array's base internal format
    is DSDT_MAG_NV and the format parameter is not DSDT_MAG_NV.

    INVALID_OPERATION is generated when TexSubImage2D
    or TexSubImage1D is called and the texture array's base internal
    format is DSDT_MAG_INTENSITY_NV and the format parameter is not
    DSDT_MAG_VIRBANCE_NV.

    INVALID_OPERATION is generated when TexEnv is called and the
    PREVIOUS_TEXTURE_INPUT_NV parameter for texture unit i is assigned
    the value TEXTUREi_ARB where f i is greater than or equal to the
    current active texture unit.

    INVALID_OPERATION is generated when TexEnv is called and the
    SHADER_OPERATION_NV parameter for texture unit 0 is assigned
    one of OFFSET_TEXTURE_2D_NV, OFFSET_TEXTURE_2D_SCALE_NV,
    OFFSET_TEXTURE_RECTANGLE_NV, OFFSET_TEXTURE_RECTANGLE_SCALE_NV,
    DEPENDENT_AR_TEXTURE_2D_NV, DEPENDENT_GB_TEXTURE_2D_NV,
    DOT_PRODUCT_NV, DOT_PRODUCT_DEPTH_REPLACE_NV,
    DOT_PRODUCT_TEXTURE_2D_NV, DOT_PRODUCT_TEXTURE_RECTANGLE_NV,
    DOT_PRODUCT_TEXTURE_CUBE_MAP_NV,
    DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV, DOT_PRODUCT_REFLECT_CUBE_MAP_NV.
    or DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV.

    INVALID_OPERATION is generated when TexEnv is called
    and the SHADER_OPERATION_NV parameter for texture
    unit 1 is assigned one of DOT_PRODUCT_DEPTH_REPLACE_NV,
    DOT_PRODUCT_TEXTURE_2D_NV, DOT_PRODUCT_TEXTURE_RECTANGLE_NV,
    DOT_PRODUCT_TEXTURE_CUBE_MAP_NV,
    DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV, DOT_PRODUCT_REFLECT_CUBE_MAP_NV,
    or DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV.

    INVALID_OPERATION is generated when TexEnv is called
    and the SHADER_OPERATION_NV parameter for texture
    unit 2 is assigned one of
    DOT_PRODUCT_TEXTURE_CUBE_MAP_NV, DOT_PRODUCT_REFLECT_CUBE_MAP_NV,
    or DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV.

    INVALID_OPERATION is generated when TexEnv is called and the
    SHADER_OPERATION_NV parameter for texture unit n-1 (where n is the
    number of supported texture units) is assigned either DOT_PRODUCT_NV
    or DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV.

    INVALID_OPERATION is generated when GetTexImage is called with a
    color format (one of RED, GREEN, BLUE, ALPHA, RGB, RGBA, BGR, BGRA,
    LUMINANCE, or LUMINANCE_ALPHA) when the texture image is of a format
    type (see table 3.15) other than RGBA (the DSDT_MAG_INTENSITY_NV
    base internal format does not count as an RGBA format type in this
    context).
```

    INVALID_OPERATION is generated when GetTexImage is called with a
    format of HILO_NV when the texture image is of a format type (see
    table 3.15) other than HILO_NV.

    INVALID_OPERATION is generated when GetTexImage is called with a
    format of DSDT_NV when the texture image is of a base internal
    format other than DSDT_NV.

    INVALID_OPERATION is generated when GetTexImage is called with a
    format of DSDT_MAG_NV when the texture image is of a base internal
    format other than DSDT_MAG_NV.

    INVALID_OPERATION is generated when GetTexImage is called with a
    format of DSDT_MAG_VIBRANCE_NV when the texture image is of a base
    internal format other than DSDT_MAG_INTENSITY_NV causes the error
    INVALID_OPERATION."

New State

Add the following entries to table 6.12:

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
TEXTURE_HI_SIZE_NV	nxZ+	GetTexLevelParameter	0	xD texture image i's hi resolution	3.8	texture
TEXTURE_LO_SIZE_NV	nxZ+	GetTexLevelParameter	0	xD texture image i's lo resolution	3.8	texture
TEXTURE_DS_SIZE_NV	nxZ+	GetTexLevelParameter	0	xD texture image i's ds resolution	3.8	texture
TEXTURE_DT_SIZE_NV	nxZ+	GetTexLevelParameter	0	xD texture image i's dt resolution	3.8	texture
TEXTURE_MAG_SIZE_NV	nxZ+	GetTexLevelParameter	0	xD texture image i's mag resolution	3.8	texture

Change the TEXTURE_BORDER_COLOR line in table 6.13 to read:

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
TEXTURE_BORDER_VALUES_NV (TEXTURE_BORDER_COLOR)	4xR	GetTexParameter	(0,0,0,0)	Texture border values	3.8	texture

Table 6.TextureShaders. Texture Shaders.

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
HI_BIAS_NV	R	GetFloatv	0.0	Hi bias for HILO	3.6.3	pixel
LO_BIAS_NV	R	GetFloatv	0.0	Lo bias for HILO	3.6.3	pixel
DS_BIAS_NV	R	GetFloatv	0.0	Ds bias	3.6.3	pixel
DT_BIAS_NV	R	GetFloatv	0.0	Dt bias	3.6.3	pixel
MAGNITUDE_BIAS_NV	R	GetFloatv	0.0	Magnitude bias	3.6.3	pixel
VIBRANCE_BIAS_NV	R	GetFloatv	0.0	Vibrance bias	3.6.3	pixel
HI_SCALE_NV	R	GetFloatv	1.0	Hi scale	3.6.3	pixel
LO_SCALE_NV	R	GetFloatv	1.0	Lo scale	3.6.3	pixel
DS_SCALE_NV	R	GetFloatv	1.0	Ds scale	3.6.3	pixel
DT_SCALE_NV	R	GetFloatv	1.0	Dt scale	3.6.3	pixel
MAGNITUDE_SCALE_NV	R	GetFloatv	1.0	Magnitude scale	3.6.3	pixel
VIBRANCE_SCALE_NV	R	GetFloatv	1.0	Vibrance scale	3.6.3	pixel
TEXTURE_SHADER_NV	B	IsEnabled	False	Texture shaders enable	3.8	texture/enable
SHADER_OPERATION_NV	TxZ21	GetTexEnviv	NONE	Texture shader operation	3.8.13	texture
CULL_MODES_NV	Tx4xZ2	GetTexEnviv	GEQUAL,GEQUAL, GEQUAL,GEQUAL	Texture shader cull fragment modes	3.8.13	texture
RGBA_UNSIGNED_-DOT_PRODUCT_MAPPING_NV	TxZ2	GetTexEnviv	UNSIGNED_IDENTITY_NV	Texture shader RGBA dot product mapping	3.8.13	texture
PREVIOUS_TEXTURE_INPUT_NV	TxZn	GetTexEnviv	TEXTURE0_ARB	Texture shader previous tex input	3.8.13	texture
CONST_EYE_NV	TxRx3	GetTexEnvfv	(0,0,-1)	Shader constant eye vector	3.8.13	texture
OFFSET_TEXTURE_MATRIX_NV	TxM2	GetTexEnvfv	(1,0,0,1)	2x2 texture offset matrix	3.8.13	texture
OFFSET_TEXTURE_SCALE_NV	TxR	GetTexEnvfv	1	Texture offset scale	3.8.13	texture
OFFSET_TEXTURE_BIAS_NV	TxR	GetTexEnvfv	0	Texture offset bias	3.8.13	texture
SHADER_CONSISTENT_NV	TxB	GetTexEnviv	True	Texture shader stage consistency	3.8.13	texture

[The "Tx" type prefix means that the state is per-texture unit.]

[The "Zn" type is an n-valued integer where n is the implementation-dependent number of texture units supported.]

New Implementation State

None

Revision History

March 29, 2001 - document that using signed HILO with a dot product shader forces the square root to zero if the 1.0-HI*HI-LO*LO value is negative.

November 15, 2001 - document that depth replace is after polygon offset; add polygon offset issue and multisample issue.

November 26, 2001 - Properly document the various TEXTURE_*_SIZE_NV texture resolution query tokens. Add table 6.12 entries.

June 5, 2002 - Driver implementations before this date incorrectly swap the HI and LO components when specifying GL_TEXTURE_BORDER_VALUES_NV when rendering via hardware. Drivers after this date have fixed the problem and match the specified behavior.

July 2, 2003 - CULL_MODES_NV, OFFSET_TEXTURE_MATRIX_NV, OFFSET_TEXTURE_2D_MATRIX_NV, and CONST_EYE_NV should not be specified to work with glTexEnvi & glTexEnvf (they can only be used with glTexEnviv & glTexEnvfv).

October 19, 2006 - Add interaction with ARB_color_buffer_float to document how ths extension behaves when ARB_color_buffer_float is also supported and when its CLAMP_FRAGMENT_COLOR_ARB state is either FIXED_ONLY_ARB when rendering to a floating-point color framebuffer or FALSE.

March 13, 2007 - Fix OFFSET_TEXTURE_2D_SCALE_NV operation to clamp the scale factor to [0,1] before multiplying it by red, green, and blue to match the hardware's actual behavior.

Name

    NV_texture_shader2

Name Strings

    GL_NV_texture_shader2

Notice

IP Status

    NVIDIA Proprietary.

Version

    NVIDIA Date: April 29, 2004
    $Id: //sw/main/docs/OpenGL/specs/GL_NV_texture_shader2.txt#9 $

Number

    231

Dependencies

    Written based on the wording of the OpenGL 1.2.1 specification,
    augmented by the NV_texture_shader extension specification.

    Requires support for the NV_texture_shader extension.

Overview

    This extension extends the NV_texture_shader functionality to
    support texture shader operations for 3D textures.

    See the NV_texture_shader extension for information about the
    texture shader operational model.

    The two new texture shader operations are:

    <conventional textures>

    22. TEXTURE_3D - Accesses a 3D texture via (s/q,t/q,r/q).

    <dot product textures>

    23. DOT_PRODUCT_TEXTURE_3D_NV - When preceded by two DOT_PRODUCT_NV
         programs in the previous two texture shader stages, computes a
         third similar dot product and composes the three dot products
         into (s,t,r) texture coordinate set to access a 3D non-projective
         texture.

Issues

Why a separate extension?

Not all implementations of NV_texture_shader will support 3D textures in hardware.

Breaking this extension out into a distinct extension allows OpenGL programs that only would use 3D textures if they are supported in hardware to determine whether hardware support is available by explicitly looking for the NV_texture_shader2 extension.

What if an implementation wanted to support NV_texture_shader2 operations within a software rasterizer?

Implementations should be free to implement the 3D texture texture shader operations in software. In this case, the implementation should NOT advertise the NV_texture_shader2 extension, but should still accept the GL_TEXTURE_3D and GL_DOT_PRODUCT_TEXTURE_3D_NV texture shader operations without an error. Likewise, the glTexImage3D command should accept the new internal texture formats, formats, and types allowed by this extension should be accepted without an error.

When NV_texture_shader2 is not advertised in the GL_EXTENSIONS string, but the extension functionality works without GL errors, programs should expect that these two texture shader operations are slow.

New Procedures and Functions

None.

New Tokens

When the <target> and <pname> parameters of TexEnvf, TexEnvfv, TexEnvi, and TexEnviv are TEXTURE_SHADER_NV and SHADER_OPERATION_NV respectively, then the value of <param> or the value pointed to by <params> may be:

```
    TEXTURE_3D
    DOT_PRODUCT_TEXTURE_3D_NV                0x86EF
```

Accepted by the <format> parameter of TexImage3D and TexSubImage3D:

```
    HILO_NV                                  0x86F4
    DSDT_NV                                  0x86F5
    DSDT_MAG_NV                              0x86F6
    DSDT_MAG_VIB_NV                          0x86F7
```

Accepted by the <type> parameter of TexImage3D and TexSubImage3D:

```
    UNSIGNED_INT_S8_S8_8_8_NV                0x86DA
    UNSIGNED_INT_8_8_S8_S8_REV_NV            0x86DB
```

Accepted by the <internalformat> parameter of TexImage3D:

SIGNED_RGBA_NV	0x86FB
SIGNED_RGBA8_NV	0x86FC
SIGNED_RGB_NV	0x86FE
SIGNED_RGB8_NV	0x86FF
SIGNED_LUMINANCE_NV	0x8701
SIGNED_LUMINANCE8_NV	0x8702
SIGNED_LUMINANCE_ALPHA_NV	0x8703
SIGNED_LUMINANCE8_ALPHA8_NV	0x8704
SIGNED_ALPHA_NV	0x8705
SIGNED_ALPHA8_NV	0x8706
SIGNED_INTENSITY_NV	0x8707
SIGNED_INTENSITY8_NV	0x8708
SIGNED_RGB_UNSIGNED_ALPHA_NV	0x870C
SIGNED_RGB8_UNSIGNED_ALPHA8_NV	0x870D

Accepted by the <internalformat> parameter of TexImage3D:

HILO_NV	
HILO16_NV	0x86F8
SIGNED_HILO_NV	0x86F9
SIGNED_HILO16_NV	0x86FA
DSDT_NV	
DSDT8_NV	0x8709
DSDT_MAG_NV	
DSDT8_MAG8_NV	0x870A
DSDT_MAG_INTENSITY_NV	0x86DC
DSDT8_MAG8_INTENSITY8_NV	0x870B

Additions to Chapter 2 of the 1.2 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.2 Specification (Rasterization)

-- Section 3.8 "Texturing"

Replace the third paragraph (amended by the NV_texture_shader extension) with the following that includes 3D texture references:

"The alternative to conventional texturing is the texture shaders mechanism. When texture shaders are enabled, each texture unit uses one of twenty-three texture shader operations. Twenty of the twenty-three shader operations map an (s,t,r,q) texture coordinate set to an RGBA color. Of these, four texture shader operations directly correspond to the 1D, 2D, 3D, and cube map conventional texturing operations. Depending on the texture shader operation, the mapping from the (s,t,r,q) texture coordinate set to an RGBA color may depend on the given texture unit's currently bound texture object state and/or the results of previous texture shader operations. The three remaining texture shader operations respectively provide a fragment culling mechanism based on texture coordinates, a means to replace the fragment depth value, and a dot product operation that computes a floating-point value for use by subsequent texture shaders. The specifics of each texture shader operation are described in section 3.8.12."

-- Section 3.8.2 "Alternate Texture Image Specification Commands"

Amend the following text inserted by NV_texture_shader after the six paragraph to include 3D texture references:

"CopyTexSubImage3D, CopyTexSubImage2D, and CopyTexSubImage1D generate the error INVALID_OPERATION if the internal format of the texture array to which the pixels are to be copied is one of HILO_NV, HILO16_NV, SIGNED_HILO_NV, SIGNED_HILO16_NV, DSDT_NV, DSDT8_NV, DSDT_MAG_NV, DSDT8_MAG8_NV, DSDT_MAG_INTENSITY_NV, or DSDT8_MAG8_INTENSITY8_NV.

TexSubImage3D, TexSubImage2D, and TexSubImage1D generate the error INVALID_OPERATION if the internal format of the texture array to which the texels are to be copied has a different format type (according to table 3.15) than the format type of the texels being specified. Specifically, if the base internal format is not one of HILO_NV, DSDT_NV, DSDT_MAG_NV, or DSDT_INTENSITY_NV, then the format parameter must be one of COLOR_INDEX, RED, GREEN, BLUE, ALPHA, RGB, RGBA, LUMINANCE, or LUMINANCE_ALPHA; if the base internal format is HILO_NV, then the format parameter must be HILO_NV; if the base internal format is DSDT_NV, then the format parameter must be DSDT_NV; if the base internal format is DSDT_MAG_NV, then the format parameter must be DSDT_MAG_NV; if the base internal format is DSDT_MAG_INTENSITY_NV, the format parameter must be DSDT_MAG_VIB_NV."

-- Section 3.8.13 "Texture Shaders"

Amend the designated paragraphs of the NV_texture_shader specification to include discussion of 3D textures.

1st paragraph:

"Each texture unit is configured with one of twenty-three texture shader operations. Several texture shader operations require additional state. All per-texture shader stage state is specified using the TexEnv commands with the target specified as TEXTURE_SHADER_NV. The per-texture shader state is replicated per texture unit so the texture unit selected by ActiveTextureARB determines which texture unit's environment is modified by TexEnv calls."

3rd paragraph:

"When TexEnv is called with the target of TEXTURE_SHADER_NV, SHADER_OPERATION_NV may be set to one of NONE, TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP_ARB, PASS_THROUGH_NV, CULL_FRAGMENT_NV, OFFSET_TEXTURE_2D_NV, OFFSET_TEXTURE_2D_SCALE_NV, OFFSET_TEXTURE_RECTANGLE_NV, OFFSET_TEXTURE_RECTANGLE_SCALE_NV, DEPENDENT_AR_TEXTURE_2D_NV, DEPENDENT_GB_TEXTURE_2D_NV, DOT_PRODUCT_NV, DOT_PRODUCT_DEPTH_REPLACE_NV, DOT_PRODUCT_TEXTURE_2D_NV, DOT_PRODUCT_TEXTURE_RECTANGLE_NV, DOT_PRODUCT_TEXTURE_3D_NV, DOT_PRODUCT_TEXTURE_CUBE_MAP_NV, DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV, DOT_PRODUCT_REFLECT_CUBE_MAP_NV, or DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV. The semantics of each of these shader operations is described in section 3.8.13.1. Not every operation is supported in every texture unit. The restrictions for how these shader operations can be configured in various texture units are described in section 3.8.13.2."

3.8.13.1 Texture Shader Operations

Amend tables 3.A, 3.B, 3.C, and 3.D in the NV_texture_shader specification to include entries for 3D texture operations:

Table 3.A:

texture shader operation i	previous texture input	texture shader operation i-1	texture shader operation i-2	texture shader operation i+1
TEXTURE_3D	-	-	-	-
DOT_PRODUCT_TEXTURE_3D_NV	shader result type must be one of signed HILO, unsigned HILO, all signed RGBA, all unsigned RGBA	shader operation must be DOT_PRODUCT_NV	shader operation must be DOT_PRODUCT_NV	-

Table 3.B:

texture shader operation i	texture unit i
TEXTURE_3D	3D target must be consistent
DOT_PRODUCT_TEXTURE_3D_NV	3D target must be consistent

Table 3.C:

texture shader operation i	texture coordinate set usage	texture target	uses stage result i-1	uses stage result i-2	uses stage result i+1	uses previous texture input	uses cull modes	uses offset texture 2D matrix	offset texture 2D scale and bias	uses const eye vector
TEXTURE_3D	s,t,r,q	3D	-	-	-	-	-	-	-	-
DOT_PRODUCT_TEXTURE_3D_NV	s,t,r	3D	y	y	-	y	-	-	-	-

Table 3.D:

texture shader operation i	shader stage result type	shader stage result	texture unit RGBA color result
TEXTURE_3D	matches 3D target type	filtered 3D target texel	if 3D target texture type is RGBA, filtered 3D target texel, else (0,0,0,0)
DOT_PRODUCT_TEXTURE_3D_NV	matches 3D target type	filtered 3D target texel	if 3D target texture type is RGBA, filtered 3D target texel, else (0,0,0,0)

Add the following new sections specifying new 3D texture operations:

3.8.13.1.22 3D Projective Texturing

The TEXTURE_3D texture shader operation accesses the texture unit's

3D texture object (as described in sections 3.8.4, 3.8.5, and 3.8.6) using (s/q,t/q,r/q) for the 3D texture coordinates where s, t, r, and q are the homogeneous texture coordinates for the texture unit. The result of the texture access becomes both the shader result and texture unit RGBA result (see table 3.E). The type of the shader result depends on the format type of the accessed texture. This mode is equivalent to conventional texturing's 3D texture target.

If the texture unit's 3D texture object is not consistent, then this texture shader stage is not consistent.

If this texture shader stage is not consistent, it operates as if it is the NONE operation.

3.8.13.1.23 Dot Product Texture 3D

The DOT_PRODUCT_TEXTURE_3D_NV texture shader operation accesses the texture unit's 3D texture object (as described in sections 3.8.4, 3.8.5, and 3.8.6) using (dotPP,dotP,dotC) for the 3D texture coordinates. The result of the texture access becomes both the shader result and texture unit RGBA result (see table 3.E). The type of the shader result depends on the format type of the accessed texture.

Assuming that i is the current texture shader stage, dotPP is the floating-point dot product texture shader result from the i-2 texture shader stage, assuming the i-2 texture shader stage's operation is DOT_PRODUCT_NV. dotP is the floating-point dot product texture shader result from the i-1 texture shader stage, assuming the i-1 texture shader stage's operation is DOT_PRODUCT_NV. dotC is the floating-point dot product result from the current texture shader stage. dotC is computed in the identical manner used to compute the floating-point result of the DOT_PRODUCT_NV texture shader described in section 3.8.13.1.14.

If the previous texture input texture object specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value has a format type other than RGBA or HILO (the DSDT_MAG_INTENSITY_NV base internal format does not count as an RGBA format type in this context), then this texture shader stage is not consistent.

If the previous texture input texture shader operation specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is DOT_PRODUCT_NV, then this texture shader stage is not consistent.

If the previous texture input texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is invalid, then this texture shader stage is not consistent.

If the previous texture input shader stage specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is not consistent, then this texture shader stage is not consistent.

If either the i-1 or i-2 texture shader stage operation is not DOT_PRODUCT_NV, then this texture shader stage is not consistent.

If either the i-1 or i-2 texture shader stage is not consistent, then this texture shader stage is not consistent.

If the texture unit's 3D texture object is not consistent, then
this texture shader stage is not consistent.

If this texture shader stage is not consistent, it operates as if
it is the NONE operation.

3.8.13.2 Texture Shader Restrictions

Amend the first four paragraphs in this section to include 3D
texture operations:

"There are various restrictions on possible texture shader
configurations. These restrictions are described in this section.

The error INVALID_OPERATION occurs if the SHADER_OPERATION_NV
parameter for texture unit 0 is assigned one of
OFFSET_TEXTURE_2D_NV, OFFSET_TEXTURE_2D_SCALE_NV,
OFFSET_TEXTURE_RECTANGLE_NV, OFFSET_TEXTURE_RECTANGLE_SCALE_NV,
DEPENDENT_AR_TEXTURE_2D_NV, DEPENDENT_GB_TEXTURE_2D_NV,
DOT_PRODUCT_NV, DOT_PRODUCT_DEPTH_REPLACE_NV,
DOT_PRODUCT_TEXTURE_2D_NV, DOT_PRODUCT_TEXTURE_RECTANGLE_NV,
DOT_PRODUCT_TEXTURE_3D_NV, DOT_PRODUCT_TEXTURE_CUBE_MAP_NV,
DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV, DOT_PRODUCT_REFLECT_CUBE_MAP_NV,
or DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV. Each of these
texture shaders requires a previous texture shader result that
is not possible for texture unit 0. Therefore these shaders are
disallowed for texture unit 0.

The error INVALID_OPERATION occurs if the
SHADER_OPERATION_NV parameter for texture unit
1 is assigned one of DOT_PRODUCT_DEPTH_REPLACE_NV,
DOT_PRODUCT_TEXTURE_2D_NV, DOT_PRODUCT_TEXTURE_RECTANGLE_NV,
DOT_PRODUCT_TEXTURE_3D_NV, DOT_PRODUCT_TEXTURE_CUBE_MAP_NV,
DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV, DOT_PRODUCT_REFLECT_CUBE_MAP_NV,
or DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV. Each of these texture
shaders requires either two previous texture shader results or
a dot product result that cannot be generated by texture unit 0.
Therefore these shaders are disallowed for texture unit 1.

The error INVALID_OPERATION occurs if the
SHADER_OPERATION_NV parameter for texture unit
2 is assigned one of DOT_PRODUCT_TEXTURE_3D_NV,
DOT_PRODUCT_TEXTURE_CUBE_MAP_NV, DOT_PRODUCT_REFLECT_CUBE_MAP_NV,
DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV. Each of these texture
shaders requires three previous texture shader results. Therefore
these shaders are disallowed for texture unit 2."

3.8.13.3 Required State

Amend the first paragraph in this section to account for the 2 new
3D texture shader operations:

"The state required for texture shaders consists of a single bit to
indicate whether or not texture shaders are enabled, a vector of
three floating-point values for the constant eye vector, and n sets

of per-texture unit state where n is the implementation-dependent number of supported texture units. The set of per-texture unit texture shader state consists of the twenty-three-valued integer indicating the texture shader operation, four two-valued integers indicating the cull modes, an integer indicating the previous texture unit input, a two-valued integer indicating the RGBA unsigned dot product mapping mode, a 2x2 floating-point matrix indicating the texture offset transform, a floating-point value indicating the texture offset scale, a floating-point value indicating the texture offset bias, and a bit to indicate whether or not the texture shader stage is consistent."

Additions to Chapter 4 of the 1.2 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.2 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

None

Additions to the GLX Specification

None

Dependencies on other specifications

Same as the NV_texture_shader extension.

Errors

The following errors are updated to reflect 3D texture operations:

INVALID_OPERATION is generated if a packed pixel format type listed in table 3.8 is used with DrawPixels, ReadPixels, ColorTable, ColorSubTable, ConvolutionFilter1D, ConvolutionFilter2D, SeparableFilter2D, GetColorTable, GetConvolutionFilter, GetSeparableFilter, GetHistogram, GetMinmax, TexImage1D, TexImage2D, TexImage3D, TexSubImage1D, TexSubImage2D, TexSubImage3d, or GetTexImage but the format parameter does not match on of the allowed Matching Pixel Formats listed in table 3.8 for the specified packed type parameter.

INVALID_OPERATION is generated when TexImage1D, TexImage2D, or TexImage3D are called and the format is HILO_NV and the internalformat is not one of HILO_NV, HILO16_NV, SIGNED_HILO_NV, SIGNED_HILO16_NV; or if the internalformat is one of HILO_NV, HILO16_NV, SIGNED_HILO_NV, or SIGNED_HILO16_NV and the format is not HILO_NV.

INVALID_OPERATION is generated when TexImage3D, TexImage2D, or TexImage1D is called and if the format is DSDT_NV and the internalformat is not either DSDT_NV or DSDT8_NV; or if the internal

```
    format is either DSDT_NV or DSDT8_NV and the format is not DSDT_NV.

    INVALID_OPERATION is generated when TexImage3D, TexImage2D, or
    TexImage1D is called and if the format is DSDT_MAG_NV and the
    internalformat is not either DSDT_MAG_NV or DSDT8_MAG8_NV; or if
    the internal format is either DSDT_MAG_NV or DSDT8_MAG8_NV and the
    format is not DSDT_MAG_NV.

    INVALID_OPERATION is generated when TexImage3D, TexImage2D,
    or TexImage1D is called and if the format is DSDT_MAG_VIB_NV
    and the internalformat is not either DSDT_MAG_INTENSITY_NV or
    DSDT8_MAG8_INTENSITY8_NV; or if the internal format is either
    DSDT_MAG_INTENSITY_NV or DSDT8_MAG8_INTENSITY8_NV and the format
    is not DSDT_MAG_VIB_NV.

    INVALID_OPERATION is generated when CopyTexImage2D, CopyTexImage1D,
    CopyTexSubImage3D, CopyTexSubImage2D, or CopyTexSubImage1D is called
    and the internal format of the texture array to which the pixels
    are to be copied is one of HILO_NV, HILO16_NV, SIGNED_HILO_NV,
    SIGNED_HILO16_NV, DSDT_NV, DSDT8_NV, DSDT_MAG_NV, DSDT8_MAG8_NV,
    DSDT_MAG_INTENSITY_NV, or DSDT8_MAG8_INTENSITY8_NV.

    INVALID_OPERATION is generated when TexSubImage3D, TexSubImage2D, or
    TexSubImage1D is called and the texture array's base internal format
    is not one of HILO_NV, DSDT_NV, DSDT_MAG_NV, or DSDT_INTENSITY_NV,
    and the format parameter is not one of COLOR_INDEX, RED,
    GREEN, BLUE, ALPHA, RGB, RGBA, LUMINANCE, or
    LUMINANCE_ALPHA

    INVALID_OPERATION is generated when TexSubImage3D, TexSubImage2D, or
    TexSubImage1D is called and the texture array's base internal format
    is HILO_NV and the format parameter is not HILO_NV.

    INVALID_OPERATION is generated when TexSubImage3D, TexSubImage2D, or
    TexSubImage1D is called and the texture array's base internal format
    is DSDT_NV and the format parameter is not DSDT_NV.

    INVALID_OPERATION is generated when TexSubImage3D, TexSubImage2D, or
    TexSubImage1D is called and the texture array's base internal format
    is DSDT_MAG_NV and the format parameter is not DSDT_MAG_NV.

    INVALID_OPERATION is generated when TexSubImage3D, TexSubImage2D,
    or TexSubImage1D is called and the texture array's base internal
    format is DSDT_MAG_INTENSITY_NV and the format parameter is not
    DSDT_MAG_VIRBANCE_NV.

    INVALID_OPERATION is generated when TexEnv is called and the
    SHADER_OPERATION_NV parameter for texture unit 0 is assigned
    one of OFFSET_TEXTURE_2D_NV, OFFSET_TEXTURE_2D_SCALE_NV,
    OFFSET_TEXTURE_RECTANGLE_NV, OFFSET_TEXTURE_RECTANGLE_SCALE_NV,
    DEPENDENT_AR_TEXTURE_2D_NV, DEPENDENT_GB_TEXTURE_2D_NV,
    DOT_PRODUCT_NV, DOT_PRODUCT_DEPTH_REPLACE_NV,
    DOT_PRODUCT_TEXTURE_2D_NV, DOT_PRODUCT_TEXTURE_RECTANGLE_NV,
    DOT_PRODUCT_TEXTURE_3D_NV, DOT_PRODUCT_TEXTURE_CUBE_MAP_NV,
    DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV, DOT_PRODUCT_REFLECT_CUBE_MAP_NV.
    or DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV.
    INVALID_OPERATION is generated when TexEnv is called
    and the SHADER_OPERATION_NV parameter for texture
    unit 1 is assigned one of DOT_PRODUCT_DEPTH_REPLACE_NV,
    DOT_PRODUCT_TEXTURE_2D_NV, DOT_PRODUCT_TEXTURE_RECTANGLE_NV,
    DOT_PRODUCT_TEXTURE_3D_NV, DOT_PRODUCT_TEXTURE_CUBE_MAP_NV,
    DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV, DOT_PRODUCT_REFLECT_CUBE_MAP_NV,
    or DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV.

    INVALID_OPERATION is generated when TexEnv is called
    and the SHADER_OPERATION_NV parameter for texture
    unit 2 is assigned one of DOT_PRODUCT_TEXTURE_3D_NV,
    DOT_PRODUCT_TEXTURE_CUBE_MAP_NV, DOT_PRODUCT_REFLECT_CUBE_MAP_NV,
    or DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV.

    INVALID_OPERATION is generated when TexEnv is called and the
    SHADER_OPERATION_NV parameter for texture unit n-1 (where n is the
    number of supported texture units) is assigned either DOT_PRODUCT_NV
    or DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV.

    INVALID_OPERATION is generated when GetTexImage is called with a
    color format (one of RED, GREEN, BLUE, ALPHA, RGB, RGBA, LUMINANCE,
    or LUMINANCE_ALPHA) when the texture image is of a format type (see
    table 3.15) other than RGBA (the DSDT_MAG_INTENSITY_NV base internal
    format does not count as an RGBA format type in this context).

    INVALID_OPERATION is generated when GetTexImage is called with
    a format of HILO when the texture image is of a format type (see
    table 3.15) other than HILO.

    INVALID_OPERATION is generated when GetTexImage is called with a
    format of DSDT_NV when the texture image is of a base internal
    format other than DSDT_NV.

    INVALID_OPERATION is generated when GetTexImage is called with a
    format of DSDT_MAG_NV when the texture image is of a base internal
    format other than DSDT_MAG_NV.

    INVALID_OPERATION is generated when GetTexImage is called with a
    format of DSDT_MAG_VIBRANCE_NV when the texture image is of a base
    internal format other than DSDT_MAG_INTENSITY_NV causes the error
    INVALID_OPERATION."
```

New State

Table 6.TextureShaders. Texture Shaders.

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
SHADER_OPERATION_NV	TxZ23	GetTexEnviv	NONE	Texture shader operation	3.8.13	texture

```
* Z21 in NV_texture_shader is now Z23 with NV_texture_shader2.

[ The "Tx" type prefix means that the state is per-texture unit. ]

[ The "Zn" type is an n-valued integer where n is the
  implementation-dependent number of texture units supported.]
```

New Implementation State

    None

Revision History

    None

Name

    NV_texture_shader3

Name Strings

    GL_NV_texture_shader3

Notice

    Copyright NVIDIA Corporation, 2001.

IP Status

    NVIDIA Proprietary.

Version

    NVIDIA Date: March 5, 2007
    $Id: //sw/main/docs/OpenGL/specs/GL_NV_texture_shader3.txt#11 $

Number

    265

Dependencies

    Written based on the wording of the OpenGL 1.2.1 specification,
    augmented by the NV_texture_shader and NV_texture_shader2 extension
    specifications.

    Requires support for the NV_texture_shader extension.

    Requires support for the NV_texture_shader2 extension.

Overview

    NV_texture_shader3 extends the NV_texture_shader functionality by
    adding several new texture shader operations, extending several
    existing texture shader operations, adding a new HILO8 internal
    format, and adding new and more flexible re-mapping modes for dot
    product and dependent texture shader operations.

    See the NV_texture_shader extension for information about the
    texture shader operational model.

    The fourteen new texture shader operations are:

    <offset textures>

    24. OFFSET_PROJECTIVE_TEXTURE_2D_NV - Transforms the signed (ds,dt)
         components of a previous texture unit by a 2x2 floating-point
         matrix and then uses the result to offset the stage's texture
         coordinates for a 2D non-projective texture.

    25. OFFSET_PROJECTIVE_TEXTURE_2D_SCALE_NV - Same as above except
         the magnitude component of the previous texture unit result
         scales the red, green, and blue components of the unsigned RGBA
         texture 2D access.

    26. OFFSET_PROJECTIVE_TEXTURE_RECTANGLE_NV - Similar to
         OFFSET_TEXTURE_2D_NV except that the texture access is into a
         rectangular non-projective texture.

27. OFFSET_PROJECTIVE_TEXTURE_RECTANGLE_SCALE_NV - Similar to OFFSET_PROJECTIVE_TEXTURE_2D_SCALE_NV except that the texture access is into a rectangular non-projective texture.

28. OFFSET_HILO_TEXTURE_2D_NV - Similar to OFFSET_TEXTURE_2D_NV but uses a (higher-precision) HILO base format texture rather than a DSDT-type base format.

29. OFFSET_HILO_TEXTURE_RECTANGLE_NV - Similar to OFFSET_TEXTURE_RECTANGLE_NV but uses a (higher-precision) HILO base format texture rather than a DSDT-type base format.

30. OFFSET_HILO_PROJECTIVE_TEXTURE_2D_NV - Similar to OFFSET_PROJECTIVE_TEXTURE_2D_NV but uses a (higher-precision) HILO base format texture rather than a DSDT-type base format.

31. OFFSET_HILO_PROJECTIVE_TEXTURE_RECTANGLE_NV - Similar to OFFSET_PROJECTIVE_TEXTURE_RECTANGLE_NV but uses a (higher-precision) HILO base format texture rather than a DSDT-type base format.

    (There are no "offset HILO texture scale" operations because HILO textures have only two components with no third component for scaling.)

<dependent textures>

32. DEPENDENT_HILO_TEXTURE_2D_NV - Converts the hi and lo components of a previous shader HILO result into an (s,t) texture coordinate set to access a 2D non-projective texture.

33. DEPENDENT_RGB_TEXTURE_3D_NV - Converts the red, green, and blue components of a previous shader RGBA result into an (s,t,r) texture coordinate set to access a 3D non-projective texture.

34. DEPENDENT_RGB_TEXTURE_CUBE_MAP_NV - Converts the red, green, and blue components of a previous shader RGBA result into an (s,t,r) texture coordinate set to access a cube map texture.

<dot product pass through>

35. DOT_PRODUCT_PASS_THROUGH_NV - Computes a dot product in the manner of the DOT_PRODUCT_NV operation and the result is [0,1] clamped and smeared to generate the texture unit RGBA result.

<dot product textures>

36. DOT_PRODUCT_TEXTURE_1D_NV - Computes a dot product in the manner of the DOT_PRODUCT_NV operation and uses the result as the s texture coordinate to access a 2D non-projective texture.

<dot product depth replace>

37. DOT_PRODUCT_AFFINE_DEPTH_REPLACE_NV - Computes a dot product in the manner of the DOT_PRODUCT_NV operation and the result is [0,1] clamped and replaces the fragment's window-space depth value. The texture unit RGBA result is (0,0,0,0).

Two new internal texture formats have been added: HILO8_NV and SIGNED_HILO8_NV. These texture formats allow HILO textures to be stored in half the space; still the filtering for these internal texture formats is done with 16-bit precision.

    One new unsigned RGBA dot product mapping mode (FORCE_BLUE_TO_ONE_NV)
    forces the blue component to be 1.0 before computing a dot product.

Issues

Should a HILO8_NV internal format be added?

RESOLUTION: Yes. The HILO8_NV format allows HILO textures to take up half the space (16-bit HILO8_NV versus 32-bit HILO16_NV). Even though the texture is stored with 8-bit components, the interpolated precision can be assumed to be 16-bit.

Should we generalize existing OFFSET_TEXTURE-style operations to support HILO textures and projective texturing, or should we just add more texture shader operations?

RESOLUTION: Add more texture shader operations for each distinct configuration.

NV_texture_shader had consistency rules for OFFSET_TEXTURE operations that preclude consistency when used with HILO textures. Consistency is a defined behavior that should stay defined even with future extensions. Adding specific new texture shader operation for HILO textures avoids having to redefine the consistency rules for DSDT-using OFFSET_TEXTURE operations.

Rather than add a separate state that decides when OFFSET_TEXTURE is projective or not, we just add new operations.

New Procedures and Functions

None.

New Tokens

When the <target> and <pname> parameters of TexEnvf, TexEnvfv, TexEnvi, and TexEnviv are TEXTURE_SHADER_NV and SHADER_OPERATION_NV respectively, then the value of <param> or the value pointed to by <params> may be:

```
        OFFSET_PROJECTIVE_TEXTURE_2D_NV                 0x8850
        OFFSET_PROJECTIVE_TEXTURE_2D_SCALE_NV           0x8851
        OFFSET_PROJECTIVE_TEXTURE_RECTANGLE_NV          0x8852
        OFFSET_PROJECTIVE_TEXTURE_RECTANGLE_SCALE_NV    0x8853
        OFFSET_HILO_TEXTURE_2D_NV                       0x8854
        OFFSET_HILO_TEXTURE_RECTANGLE_NV                0x8855
        OFFSET_HILO_PROJECTIVE_TEXTURE_2D_NV            0x8856
        OFFSET_HILO_PROJECTIVE_TEXTURE_RECTANGLE_NV     0x8857
        DEPENDENT_HILO_TEXTURE_2D_NV                    0x8858
        DEPENDENT_RGB_TEXTURE_3D_NV                     0x8859
        DEPENDENT_RGB_TEXTURE_CUBE_MAP_NV               0x885A
        DOT_PRODUCT_PASS_THROUGH_NV                     0x885B
        DOT_PRODUCT_TEXTURE_1D_NV                       0x885C
        DOT_PRODUCT_AFFINE_DEPTH_REPLACE_NV             0x885D
```

Accepted by the <internalformat> parameter of TexImage1D, TexImage2D, and TexImage3D:

```
        HILO8_NV                                        0x885E
        SIGNED_HILO8_NV                                 0x885F
```

```
    When the <target> and <pname> parameters of TexEnvf,
    TexEnvfv, TexEnvi, and TexEnviv are TEXTURE_SHADER_NV and
    RGBA_UNSIGNED_DOT_PRODUCT_MAPPING_NV respectively, then the value
    of <param> or the value pointed to by <params> may be:

        FORCE_BLUE_TO_ONE_NV                          0x8860
```

Additions to Chapter 2 of the 1.2 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.2 Specification (Rasterization)

-- Section 3.8 "Texturing"

Replace the third paragraph (amended by the NV_texture_shader extension) with the following that includes new texture shader references:

"The alternative to conventional texturing is the texture shaders mechanism. When texture shaders are enabled, each texture unit uses one of thirty-seven texture shader operations. Thirty-three of the thirty-seven shader operations map an (s,t,r,q) texture coordinate set to an RGBA color. Of these, four texture shader operations directly correspond to the 1D, 2D, 3D, and cube map conventional texturing operations. Depending on the texture shader operation, the mapping from the (s,t,r,q) texture coordinate set to an RGBA color may depend on the given texture unit's currently bound texture object state and/or the results of previous texture shader operations. The four remaining texture shader operations respectively provide a fragment culling mechanism based on texture coordinates, a dot product operation that computes a floating-point value for use by subsequent texture shaders. and two means to replace the fragment depth value, The specifics of each texture shader operation are described in section 3.8.12."

-- Section 3.8.1 "Texture Image Specification"

Add two more rows to table 3.16:

Sized Internal Format	Base Internal Format	R bits	G bits	B bits	A bits	L bits	I bits	HI bits	LO bits	DS bits	DT bits	MAG bits
HILO8_NV	HILO							8	8			
SIGNED_HILO8_NV	HILO							8*	8*			

Update this paragraph inserted by NV_texture_shader before the last sentence in the fifth paragraph to read:

"The error INVALID_OPERATION is generated if the format is HILO_NV and the internalformat is not one of HILO_NV, HILO16_NV, HILO8_NV, SIGNED_HILO_NV, SIGNED_HILO16_NV, SIGNED_HILO8_NV; or if the internalformat is one of HILO_NV, HILO16_NV, HILO8_NV, SIGNED_HILO_NV, SIGNED_HILO16_NV, or SIGNED_HILO8_NV and the format is not HILO_NV.

-- Section 3.8.2 "Alternate Texture Image Specification Commands"

In the second paragraph (describing CopyTexImage2D), change the third to the last sentence (previously amended by NV_texture_shader) to:

"Parameters level, internalformat, and border are specified using the same values, with the same meanings, as the equivalent arguments of TexImage2D, except that internalformat may not be specified as 1, 2, 3, 4, HILO_NV, HILO16_NV, HILO8_NV, SIGNED_HILO_NV, SIGNED_HILO16_NV, SIGNED_HILO8_NV, DSDT_NV, DSDT8_NV, DSDT_MAG_NV, DSDT8_MAG8_NV, DSDT_MAG_INTENSITY_NV, or DSDT8_MAG8_INTENSITY8_NV."

In the third paragraph (describing CopyTexImage1D), change the second to the last sentence (previously amended by NV_texture_shader) to:

"level, internalformat, and border are specified using the same values, with the same meanings, as the equivalent arguments of TexImage1D, except that internalformat may not be specified as 1, 2, 3, 4, HILO_NV, HILO16_NV, HILO8_NV, SIGNED_HILO_NV, SIGNED_HILO16_NV, SIGNED_HILO8_NV, DSDT_NV, DSDT8_NV, DSDT_MAG_NV, DSDT8_MAG8_NV, DSDT_MAG_INTENSITY_NV, or DSDT8_MAG8_INTENSITY8_NV."

Amend the following text inserted by NV_texture_shader after the six paragraph to include the HILO8 and UNSIGNED_HILO8 internal texture formats:

"CopyTexSubImage3D, CopyTexSubImage2D, and CopyTexSubImage1D generate the error INVALID_OPERATION if the internal format of the texture array to which the pixels are to be copied is one of HILO_NV, HILO16_NV, HILO8_NV, SIGNED_HILO_NV, SIGNED_HILO16_NV, SIGNED_HILO8_NV, DSDT_NV, DSDT8_NV, DSDT_MAG_NV, DSDT8_MAG8_NV, DSDT_MAG_INTENSITY_NV, or DSDT8_MAG8_INTENSITY8_NV."

-- Section 3.8.13 "Texture Shaders"

Amend the designated paragraphs of the NV_texture_shader specification to include discussion of new texture shader operations.

1st paragraph (update number of operations):

"Each texture unit is configured with one of thirty-seven texture shader operations. Several texture shader operations require additional state. All per-texture shader stage state is specified using the TexEnv commands with the target specified as TEXTURE_SHADER_NV. The per-texture shader state is replicated per texture unit so the texture unit selected by ActiveTextureARB determines which texture unit's environment is modified by TexEnv calls."

3rd paragraph (add fourteen new texture shader operations):

"When TexEnv is called with the target of TEXTURE_SHADER_NV, SHADER_OPERATION_NV may be set to one of NONE, TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP_ARB, PASS_THROUGH_NV, CULL_FRAGMENT_NV, OFFSET_TEXTURE_2D_NV, OFFSET_TEXTURE_2D_SCALE_NV, OFFSET_TEXTURE_RECTANGLE_NV, OFFSET_TEXTURE_RECTANGLE_SCALE_NV, DEPENDENT_AR_TEXTURE_2D_NV, DEPENDENT_GB_TEXTURE_2D_NV, DOT_PRODUCT_NV, DOT_PRODUCT_DEPTH_REPLACE_NV, DOT_PRODUCT_TEXTURE_2D_NV, DOT_PRODUCT_TEXTURE_RECTANGLE_NV, DOT_PRODUCT_TEXTURE_3D_NV, DOT_PRODUCT_TEXTURE_CUBE_MAP_NV, DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV, DOT_PRODUCT_REFLECT_CUBE_MAP_NV, DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV, OFFSET_PROJECTIVE_TEXTURE_2D_NV, OFFSET_PROJECTIVE_TEXTURE_2D_SCALE_NV, OFFSET_PROJECTIVE_TEXTURE_RECTANGLE_NV, OFFSET_PROJECTIVE_TEXTURE_RECTANGLE_SCALE_NV, OFFSET_HILO_TEXTURE_2D_NV, OFFSET_HILO_TEXTURE_RECTANGLE_NV, OFFSET_HILO_PROJECTIVE_TEXTURE_2D_NV, OFFSET_HILO_PROJECTIVE_TEXTURE_RECTANGLE_NV, DEPENDENT_HILO_TEXTURE_2D_NV, DEPENDENT_RGB_TEXTURE_3D_NV, DEPENDENT_RGB_TEXTURE_CUBE_MAP_NV, DOT_PRODUCT_PASS_THROUGH_NV, DOT_PRODUCT_TEXTURE_1D_NV, or DOT_PRODUCT_AFFINE_DEPTH_REPLACE_NV. The semantics of each of these shader operations is described in section 3.8.13.1. Not every operation is supported in every texture unit. The restrictions for how these shader operations can be configured in various texture units are described in section 3.8.13.2."

5th paragraph (add FORCE_BLUE_TO_ONE_NV):

"When TexEnv is called with the target of TEXTURE_SHADER_NV, RGBA_UNSIGNED_DOT_PRODUCT_MAPPING_NV may be set to one of UNSIGNED_IDENTITY_NV, EXPAND_NORMAL_NV, or FORCE_BLUE_TO_ONE_NV. This RGBA unsigned dot product mapping mode is used by the DOT_PRODUCT_NV operation (see section 3.8.13.1.14) and other operations that compute dot products."

3.8.13.1 Texture Shader Operations

Amend tables 3.A, 3.B, 3.C, and 3.D in the NV_texture_shader specification to include these new entries:

Table 3.A:

texture shader				
texture shader operation i	previous texture input	texture shader operation i-1	operation i-2	texture shader operation i+1
OFFSET_PROJECTIVE_TEXTURE_2D_NV	base internal texture format must be one of DSDT_NV, DSDT_MAG_NV, or DSDT_MAG_INTENSITY_NV	-	-	-
OFFSET_PROJECTIVE_TEXTURE_2D_SCALE_NV	base internal texture format must be either DSDT_MAG_NV or DSDT_MAG_INTENSITY_NV	-	-	-
OFFSET_PROJECTIVE_TEXTURE_RECTANGLE_NV	base internal texture format must be one of DSDT_NV, DSDT_MAG_NV, or DSDT_MAG_INTENSITY_NV	-	-	-
OFFSET_PROJECTIVE_TEXTURE_RECTANGLE_SCALE_NV	base internal texture format must be either DSDT_MAG_NV or DSDT_MAG_INTENSITY_NV	-	-	-
OFFSET_HILO_TEXTURE_2D_NV	base internal texture format must be HILO	-	-	-
OFFSET_HILO_TEXTURE_RECTANGLE_NV	base internal texture format must be HILO	-	-	-
OFFSET_HILO_PROJECTIVE_TEXTURE_2D_NV	base internal texture format must be HILO	-	-	-
OFFSET_HILO_PROJECTIVE_TEXTURE_RECTANGLE_NV	base internal texture format must be HILO	-	-	-
DEPENDENT_HILO_TEXTURE_2D_NV	base internal texture format must be HILO	-	-	-
DEPENDENT_RGB_TEXTURE_3D_NV	shader result type must all be unsigned RGBA	-	-	-
DEPENDENT_RGB_TEXTURE_CUBE_MAP_NV	shader result type must all be RGB or RGBA (signed RGB components are allowed)	-	-	-
DOT_PRODUCT_PASS_THROUGH_NV	shader result type must be one of signed HILO, unsigned HILO, all signed RGBA, or all unsigned RGBA	-	-	-
DOT_PRODUCT_TEXTURE_1D_NV	shader result type must be one of signed HILO, unsigned HILO, all signed RGBA, or all unsigned RGBA	-	-	-
DOT_PRODUCT_AFFINE_DEPTH_REPLACE_NV	shader result type must be one of signed HILO, unsigned HILO, all signed RGBA, or all unsigned RGBA	-	-	-

Table 3.B:

texture shader operation i	texture unit i
OFFSET_PROJECTIVE_TEXTURE_2D_NV	2D target must be consistent
OFFSET_PROJECTIVE_TEXTURE_2D_SCALE_NV	2D target must be consistent and 2D texture target type must be unsigned RGBA
OFFSET_PROJECTIVE_TEXTURE_RECTANGLE_NV	rectangle target must be consistent
OFFSET_PROJECTIVE_TEXTURE_RECTANGLE_SCALE_NV	rectangle target must be consistent and rectangle texture target type must be unsigned RGBA
OFFSET_HILO_TEXTURE_2D_NV	2D target must be consistent
OFFSET_HILO_TEXTURE_RECTANGLE_NV	rectangle target must be consistent
OFFSET_HILO_PROJECTIVE_TEXTURE_2D_NV	2D target must be consistent
OFFSET_HILO_PROJECTIVE_TEXTURE_RECTANGLE_NV	rectangle target must be consistent
DEPENDENT_HILO_TEXTURE_2D_NV	2D target must be consistent
DEPENDENT_RGB_TEXTURE_3D_NV	3D target must be consistent
DEPENDENT_RGB_TEXTURE_CUBE_MAP_NV	cube map target must be consistent
DOT_PRODUCT_PASS_THROUGH_NV	-
DOT_PRODUCT_TEXTURE_1D_NV	1D target must be consistent
DOT_PRODUCT_AFFINE_DEPTH_REPLACE_NV	-

Table 3.C:

texture shader operation i	texture coordinate set usage	texture target	uses stage result i-1	uses stage result i-2	uses stage result i+1	uses previous texture input	uses cull modes	uses offset texture 2D matrix	offset texture 2D scale and bias	uses const eye vector
OFFSET_PROJECTIVE_TEXTURE_2D_NV	s,t,q	2D	-	-	-	-	-	y	-	-
OFFSET_PROJECTIVE_-TEXTURE_2D_SCALE_NV	s,t,q	2D	-	-	-	-	-	y	y	-
OFFSET_PROJECTIVE_-TEXTURE_RECTANGLE_NV	s,t,q	rectangle	-	-	-	-	-	y	-	-
OFFSET_PROJECTIVE_-TEXTURE_RECTANGLE_SCALE_NV	s,t,q	rectangle	-	-	-	-	-	y	y	-
OFFSET_HILO_TEXTURE_2D_NV	s,t	2D	-	-	-	-	-	y	-	-
OFFSET_HILO_TEXTURE_RECTANGLE_NV	s,t	rectangle	-	-	-	-	-	y	-	-
OFFSET_PROJECTIVE_-HILO_TEXTURE_2D_NV	s,t,q	2D	-	-	-	-	-	y	-	-
OFFSET_PROJECTIVE_-HILO_TEXTURE_RECTANGLE_NV	s,t,q	rectangle	-	-	-	-	-	y	-	-
DEPENDENT_HILO_TEXTURE_2D_NV	-	2D	-	-	-	y	-	-	-	-
DEPENDENT_RGB_TEXTURE_3D_NV	-	3D	-	-	-	y	-	-	-	-
DEPENDENT_RGB_TEXTURE_CUBE_MAP_NV	-	cube map	-	-	-	y	-	-	-	-
DOT_PRODUCT_PASS_THROUGH_NV	s,t,r	-	-	-	-	y	-	-	-	-
DOT_PRODUCT_TEXTURE_1D_NV	s,t,r	1D	-	-	-	y	-	-	-	-
DOT_PRODUCT_AFFINE_DEPTH_REPLACE_NV	s,t,r	-	-	-	-	y	-	-	-	-

Table 3.D:

texture shader operation i	shader stage result type	shader stage result	texture unit RGBA color result
OFFSET_TEXTURE_2D_NV	matches 2D target type	filtered 2D target texel	if 2D target texture type is RGBA, filtered 2D target texel, else (0,0,0,0)
OFFSET_TEXTURE_2D_SCALE_NV	unsigned RGBA	filtered 2D target texel	scaled filtered 2D target texel
OFFSET_TEXTURE_RECTANGLE_NV	matches rectangle target type	filtered rectangle target texel	if rectangle target texture type is RGBA, filtered rectangle target texel, else (0,0,0,0)
OFFSET_TEXTURE_RECTANGLE_SCALE_NV	unsigned RGBA	filtered rectangle target texel	scaled filtered rectangle target texel
OFFSET_PROJECTIVE_TEXTURE_2D_NV	matches 2D target type	filtered 2D target texel	if 2D target texture type is RGBA, filtered 2D target texel, else (0,0,0,0)
OFFSET_PROJECTIVE_-TEXTURE_2D_SCALE_NV	unsigned RGBA	filtered 2D target texel	scaled filtered 2D target texel
OFFSET_PROJECTIVE_-TEXTURE_RECTANGLE_NV	matches rectangle target type	filtered rectangle target texel	if rectangle target texture type is RGBA, filtered rectangle target texel, else (0,0,0,0)
OFFSET_PROJECTIVE_-TEXTURE_RECTANGLE_SCALE_NV	unsigned RGBA	filtered rectangle target texel	scaled filtered rectangle target texel
DEPENDENT_HILO_TEXTURE_2D_NV	matches 2D target type	filtered 2D target texel	if 2D target texture type is RGBA, filtered 2D target texel, else (0,0,0,0)
DEPENDENT_RGB_TEXTURE_3D_NV	matches 3D target type	filtered 3D target texel	if 3D target texture type is RGBA, filtered 3D target texel, else (0,0,0,0)
DEPENDENT_RGB_TEXTURE_CUBE_MAP_NV	matches cube map target type	filtered cube map target texel	if cube map target texture type is RGBA, filtered cube map target texel, else (0,0,0,0)
DOT_PRODUCT_PASS_THROUGH_NV	unsigned RGBA	(max(0,min(1,[s,t,r]dot[a,b,c])), max(0,min(1,[s,t,r]dot[a,b,c])), max(0,min(1,[s,t,r]dot[a,b,c])), max(0,min(1,[s,t,r]dot[a,b,c])))	(max(0,min(1,[s,t,r]dot[a,b,c])), max(0,min(1,[s,t,r]dot[a,b,c])), max(0,min(1,[s,t,r]dot[a,b,c])), max(0,min(1,[s,t,r]dot[a,b,c])))
DOT_PRODUCT_TEXTURE_1D_NV	matches 1D target type	filtered 1D target texel	if 1D target texture type is RGBA, filtered 1D target texel, else (0,0,0,0)
DOT_PRODUCT_-AFFINE_DEPTH_REPLACE_NV	unsigned RGBA	invalid	(0,0,0,0)

3.8.13.1.14 Dot Product

Add this description of FORCE_BLUE_TO_ONE_NV after the description of EXPAND_NORMAL_NV:

"When the RGBA_UNSIGNED_DOT_PRODUCT_MAPPING_NV is FORCE_BLUE_TO_ONE_NV, then the floating-point result for unsigned RGBA components is computed by

```
  result = s * Rprev + t * Gprev + r
```

where Rprev and Gprev are the (unsigned) red and green components respectively of the previous texture unit's RGBA texture shader result (the previous blue component can be assumed forced to 1.0 for the purposes of the dot product computation)."

3.8.13.1.21 Dot Product Depth Replace

Amend the paragraph meant to avoid multiple depth replaces to read:

"If any previous texture shader stage operation is DOT_PRODUCT_DEPTH_REPLACE_NV or DOT_PRODUCT_AFFINE_DEPTH_REPLACE_NV and that previous stage is consistent, then this texture shader stage is not consistent. (This eliminates the potential for two stages to each be performing a depth replace operation.)"

Add the following new sections specifying new texture shader operations:

Add the following new texture shader operation descriptions:

"3.8.13.1.24 Offset Projective Texture 2D

The OFFSET_PROJECTIVE_TEXTURE_2D_NV shader operation operates identically to the OFFSET_TEXTURE_2D_NV shader operation except that the perturbed texture coordinates s' and t' are computed with floating-point math as follows:

```
  s' = s/q + a1 * DSprev + a3 * DTprev
  t' = t/q + a2 * DSprev + a4 * DTprev
```

Note the division of s and t by the current texture shader stage's q texture coordinate.

3.8.13.1.25 Offset Projective Texture 2D Scale

The OFFSET_PROJECTIVE_TEXTURE_2D_SCALE_NV shader operation operates identically to the OFFSET_TEXTURE_2D_SCALE_NV shader operation except that the perturbed texture coordinates s' and t' are computed with floating-point math as follows:

```
  s' = s/q + a1 * DSprev + a3 * DTprev
  t' = t/q + a2 * DSprev + a4 * DTprev
```

Note the division of s and t by the current texture shader stage's q texture coordinate.

3.8.13.1.26 Offset Projective Texture Rectangle

The OFFSET_PROJECTIVE_TEXTURE_RECTANGLE_NV shader operation operates identically to the OFFSET_TEXTURE_RECTANGLE_NV shader operation except that the perturbed texture coordinates s' and t' are computed with floating-point math as follows:

```
  s' = s/q + a1 * DSprev + a3 * DTprev
  t' = t/q + a2 * DSprev + a4 * DTprev
```

Note the division of s and t by the current texture shader stage's q texture coordinate.

3.8.13.1.27 Offset Projective Texture Rectangle Scale

The OFFSET_PROJECTIVE_TEXTURE_RECTANGLE_SCALE_NV shader operation operates identically to the OFFSET_TEXTURE_RECTANGLE_SCALE_NV shader operation except that the perturbed texture coordinates s' and t' are computed with floating-point math as follows:

```
  s' = s/q + a1 * DSprev + a3 * DTprev
  t' = t/q + a2 * DSprev + a4 * DTprev
```

Note the division of s and t by the current texture shader stage's q texture coordinate.

3.8.13.1.28 Offset HILO Texture 2D

The OFFSET_HILO_TEXTURE_2D_NV texture shader operation uses the transformed result of a previous texture shader stage to perturb the current texture shader stage's (s,t) texture coordinates (without a projective division by q). The resulting perturbed texture coordinates (s',t') are used to access the texture unit's 2D texture object (as described in sections 3.8.4, 3.8.5, and 3.8.6).

The result of the texture access becomes both the shader result and texture unit RGBA result (see table 3.E). The type of the shader result depends on the format type of the accessed texture.

The perturbed texture coordinates s' and t' are computed with floating-point math as follows:

```
  s' = s + a1 * HIprev + a3 * LOprev
  t' = t + a2 * HIprev + a4 * LOprev
```

where a1, a2, a3, and a4 are the texture shader stage's OFFSET_TEXTURE_MATRIX_NV values, and HIprev and LOprev are the (signed) HI and LO components of a previous texture shader unit's texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value.

If the texture unit's 2D texture object is not consistent, then this texture shader stage is not consistent.

If the previous texture input texture object specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value has a base internalformat that is not HILO with signed components, then this texture shader stage is not consistent.

If the previous texture input texture shader operation specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is DOT_PRODUCT_NV, then this texture shader stage is not consistent.

If the previous texture input texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is invalid, then this texture shader stage is not consistent.

If the previous texture input shader stage specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is not consistent, then this texture shader stage is not consistent.

If this texture shader stage is not consistent, it operates as if it is the NONE operation.

3.8.13.1.29 Offset HILO Texture Rectangle

The OFFSET_HILO_TEXTURE_RECTANGLE_NV shader operation operates identically to the OFFSET_HILO_TEXTURE_2D_NV shader operation except that the rectangle texture target is accessed rather than the 2D texture target.

If the texture unit's rectangle texture object (rather than the 2D texture object) is not consistent, then this texture shader stage is not consistent.

3.8.13.1.30 Offset Projective HILO Texture 2D

The OFFSET_HILO_PROJECTIVE_TEXTURE_2D_NV shader operation operates identically to the OFFSET_HILO_TEXTURE_2D_NV shader operation except that the perturbed texture coordinates s' and t' are computed with floating-point math as follows:

```
  s' = s/q + a1 * HIprev + a3 * LOprev
  t' = t/q + a2 * HIprev + a4 * LOprev
```

Note the division of s and t by the current texture shader stage's q texture coordinate.

3.8.13.1.31 Offset Projective HILO Texture Rectangle

The OFFSET_HILO_PROJECTIVE_TEXTURE_RECTANGLE_NV shader operation operates identically to the OFFSET_HILO_TEXTURE_RECTANGLE_NV shader operation except that the perturbed texture coordinates s' and t' are computed with floating-point math as follows:

```
  s' = s/q + a1 * HIprev + a3 * LOprev
  t' = t/q + a2 * HIprev + a4 * LOprev
```

Note the division of s and t by the current texture shader stage's q texture coordinate.

3.8.13.1.32 Dependent HILO Texture 2D

The DEPENDENT_HILO_TEXTURE_2D_NV texture shader operation accesses the texture unit's 2D texture object (as described in section 3.8.4, 3.8.5, and 3.8.6) using (HIprev, LOprev) for the 2D texture coordinates where HIprev and LOprev are the are the hi and lo components of a previous texture input's unsigned HILO texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value. The result of the texture access becomes both the shader result and texture unit RGBA result (see table 3.E). The type of the shader result depends on the format type of the accessed texture.

If the texture unit's 2D texture object is not consistent, then this texture shader stage is not consistent.

If the previous texture input's texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value has a texture shader result type other than HILO with unsigned components, then this texture shader stage is not consistent.

If the previous texture input texture shader operation specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is DOT_PRODUCT_NV, then this texture shader stage is not consistent.

If the previous texture input texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is invalid, then this texture shader stage is not consistent.

If the previous texture input shader stage specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is not consistent, then this texture shader stage is not consistent.

If this texture shader stage is not consistent, it operates as if it is the NONE operation.

3.8.13.1.33 Dependent Texture 3D

The DEPENDENT_RGB_TEXTURE_3D_NV texture shader operation accesses the texture unit's 3D texture object (as described in section 3.8.4, 3.8.5, and 3.8.6) using (Rprev, Gprev, Bprev) for the 3D texture coordinates where Rprev, Gprev, and Bprev are the are the red, green, and blue components of a previous texture input's RGBA texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value. The result of the texture access becomes both the shader result and texture unit RGBA result (see table 3.E). The type of the shader result depends on the format type of the accessed texture.

If the texture unit's 3D texture object is not consistent, then this texture shader stage is not consistent.

If the previous texture input's texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value has a texture shader result type other than RGBA (the DSDT_MAG_INTENSITY_NV base internal format does not count as an RGBA format type in this context), then this texture shader stage is not consistent.

If the previous texture input's texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value has a texture shader result type of RGBA but any of the RGBA components are signed, then this texture shader stage is not consistent.

If the previous texture input texture shader operation specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is DOT_PRODUCT_NV, then this texture shader stage is not consistent.

If the previous texture input texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is invalid, then this texture shader stage is not consistent.

If the previous texture input shader stage specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is not consistent, then this texture shader stage is not consistent.

If this texture shader stage is not consistent, it operates as if it is the NONE operation.

3.8.13.1.34 Dependent Texture Cube Map

The DEPENDENT_RGB_TEXTURE_CUBE_MAP_NV texture shader operation accesses the texture unit's cube map texture object (as described in section 3.8.4, 3.8.5, and 3.8.6) using (s',t',r').

When the RGB components of the previous texture input's RGBA texture shader result are all unsigned, s', t', and r' are computed as:

```
  s' = 2*(Rprev - 0.5)
  t' = 2*(Gprev - 0.5)
  r' = 2*(Bprev - 0.5)
```

When the RGB components of the previous texture input's RGBA texture shader result are all signed, s', t', and r' are computed as:

```
  s' = Rprev
  t' = Gprev
  r' = Bprev
```

where Rprev, Gprev, and Bprev are the are the red, green, and blue components of a previous texture input's RGBA texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value. The result of the texture access becomes both the shader result and texture unit RGBA result (see table 3.E). The type of the shader result depends on the format type of the accessed texture.

If the texture unit's cube map texture object is not consistent, then this texture shader stage is not consistent.

If the previous texture input's texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value has a texture shader result type other than RGBA (the DSDT_MAG_INTENSITY_NV base internal format does not count as an RGBA format type in this context), then this texture shader stage is not consistent.

If the previous texture input texture shader operation specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is DOT_PRODUCT_NV, then this texture shader stage is not consistent.

If the previous texture input texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is invalid, then this texture shader stage is not consistent.

If the previous texture input shader stage specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is not consistent, then this texture shader stage is not consistent.

If this texture shader stage is not consistent, it operates as if it is the NONE operation.

3.8.13.1.35 Dot Product Pass Through

The DOT_PRODUCT_PASS_THROUGH_NV texture shader operation converts a dot product result dotC into an RGBA color result (x,x,x,x) where x is dotC clamped to [0,1]. The texture shader result and texture unit RGBA result of this operation are both assigned the clamped RGBA color result.

dotC is the floating-point dot product result from the current texture shader stage. dotC is computed in the identical manner used to compute the floating-point result of the DOT_PRODUCT_NV texture shader described in section 3.8.13.1.14.

This operation in no way depends on any of the texture unit's texture objects.

3.8.13.1.36 Dot Product Texture 1D

The DOT_PRODUCT_TEXTURE_1D_NV texture shader operation accesses the texture unit's 1D texture object (as described in sections 3.8.4, 3.8.5, and 3.8.6) using dotC for the 1D texture coordinate. The result of the texture access becomes both the shader result and texture unit RGBA result (see table 3.E). The type of the shader result depends on the format type of the accessed texture.

dotC is the floating-point dot product result from the current texture shader stage. dotC is computed in the identical manner used to compute the floating-point result of the DOT_PRODUCT_NV texture shader described in section 3.8.13.1.14.

If the previous texture input texture object specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value has a format type other than RGBA or HILO (the DSDT_MAG_INTENSITY_NV base internal format does not count as an RGBA format type in this context), then this texture shader stage is not consistent.

If the previous texture input texture shader operation specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is DOT_PRODUCT_NV, then this texture shader stage is not consistent.

If the previous texture input texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is invalid, then this texture shader stage is not consistent.

If the previous texture input shader stage specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is not consistent, then this texture shader stage is not consistent.

If the texture unit's 1D texture object is not consistent, then this texture shader stage is not consistent.

If this texture shader stage is not consistent, it operates as if it is the NONE operation.

3.8.13.1.37 Dot Product Affine Depth Replace

The DOT_PRODUCT_AFFINE_DEPTH_REPLACE_NV texture shader operation replaces the incoming fragments depth (in window coordinates, after polygon offset and before conversion to fixed-point, i.e. in the [0,1] range) with a new depth value. The new depth is computed as follows:

```
  depth = dotC
```

dotC is the floating-point dot product result from the current texture shader stage. dotC is computed in the identical manner used to compute the floating-point result of the DOT_PRODUCT_NV texture shader described in section 3.8.13.1.14. Note that there is no divide to project the depth value as is the case with the projective DOT_PRODUCT_DEPTH_REPLACE_NV operation.

If the new depth value is outside of the range of the near and far depth range values, the fragment is rejected.

The texture unit RGBA result generated is always (0,0,0,0). The texture shader result is invalid.

If the previous texture input texture object specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value has a format type other than RGBA or HILO (the DSDT_MAG_INTENSITY_NV base internal format does not count as an RGBA format type in this context), then this texture shader stage is not consistent.

If the previous texture input texture shader operation specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is DOT_PRODUCT_NV, then this texture shader stage is not consistent.

If the previous texture input texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is invalid, then this texture shader stage is not consistent.

If the previous texture input shader stage specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is not consistent, then this texture shader stage is not consistent.

If any previous texture shader stage operation is DOT_PRODUCT_DEPTH_REPLACE_NV or DOT_PRODUCT_AFFINE_DEPTH_REPLACE_NV and that previous stage is consistent, then this texture shader stage is not consistent. (This eliminates the potential for two stages to each be performing a depth replace operation.)

If this texture shader stage is not consistent, it operates as if it is the NONE operation.

This operation in no way depends on any of the texture unit's texture objects."

3.8.13.2 Texture Shader Restrictions

Amend the first two paragraphs in this section to include the new texture shader operations:

"There are various restrictions on possible texture shader configurations. These restrictions are described in this section.

The error INVALID_OPERATION occurs if the SHADER_OPERATION_NV parameter for texture unit 0 is assigned one of OFFSET_TEXTURE_2D_NV, OFFSET_TEXTURE_2D_SCALE_NV, OFFSET_TEXTURE_RECTANGLE_NV, OFFSET_TEXTURE_RECTANGLE_SCALE_NV, DEPENDENT_AR_TEXTURE_2D_NV, DEPENDENT_GB_TEXTURE_2D_NV, DOT_PRODUCT_NV, DOT_PRODUCT_DEPTH_REPLACE_NV, DOT_PRODUCT_TEXTURE_2D_NV, DOT_PRODUCT_TEXTURE_RECTANGLE_NV, DOT_PRODUCT_TEXTURE_3D_NV, DOT_PRODUCT_TEXTURE_CUBE_MAP_NV, DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV, DOT_PRODUCT_REFLECT_CUBE_MAP_NV, DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV, OFFSET_PROJECTIVE_TEXTURE_2D, OFFSET_PROJECTIVE_TEXTURE_2D_SCALE, OFFSET_PROJECTIVE_TEXTURE_RECTANGLE, OFFSET_PROJECTIVE_TEXTURE_RECTANGLE_SCALE, OFFSET_HILO_TEXTURE_2D, OFFSET_HILO_TEXTURE_RECTANGLE, OFFSET_HILO_PROJECTIVE_TEXTURE_2D, OFFSET_HILO_PROJECTIVE_TEXTURE_RECTANGLE, DEPENDENT_HILO_TEXTURE_2D, DEPENDENT_RGB_TEXTURE_3D, DEPENDENT_RGB_TEXTURE_CUBE_MAP, DOT_PRODUCT_PASS_THROUGH, DOT_PRODUCT_TEXTURE_1D, or DOT_PRODUCT_AFFINE_DEPTH_REPLACE. Each of these texture shaders requires a previous texture shader result that is not possible for texture unit 0. Therefore these shaders are disallowed for texture unit 0."

3.8.13.3 Required State

Amend the first paragraph in this section to account for the 9 new texture shader operations and the new "dot product third component" state:

"The state required for texture shaders consists of a single bit to indicate whether or not texture shaders are enabled, a vector of three floating-point values for the constant eye vector, and n sets of per-texture unit state where n is the implementation-dependent number of supported texture units. The set of per-texture unit texture shader state consists of the thirty-seven-valued integer indicating the texture shader operation, four two-valued integers indicating the cull modes, an integer indicating the previous texture unit input, a two-valued integer indicating the RGBA unsigned dot product mapping mode, a 2x2 floating-point matrix indicating the texture offset transform, a floating-point value indicating the texture offset scale, a floating-point value indicating the texture offset bias, and a bit to indicate whether or not the texture shader stage is consistent."

Additions to Chapter 4 of the 1.2 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.2 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

None

Additions to the GLX Specification

None

Dependencies on other specifications

Same as the NV_texture_shader extension.

Errors

```
INVALID_OPERATION is generated when TexImage1D, TexImage2D,
or TexImage3D are called and the format is HILO_NV and the
internalformat is not one of HILO_NV, HILO8_NV, HILO16_NV,
SIGNED_HILO_NV, SIGNED_HILO8_NV, SIGNED_HILO16_NV; or if
the internalformat is one of HILO_NV, HILO8_NV, HILO16_NV,
SIGNED_HILO_NV, SIGNED_HILO8_NV or SIGNED_HILO16_NV and the format
is not HILO_NV.

INVALID_OPERATION is generated when CopyTexImage2D, CopyTexImage1D,
CopyTexSubImage3D, CopyTexSubImage2D, or CopyTexSubImage1D is called
and the internal format of the texture array to which the pixels are
to be copied is one of HILO_NV, HILO8_NV, HILO16_NV, SIGNED_HILO_NV,
SIGNED_HILO8_NV, SIGNED_HILO16_NV, DSDT_NV, DSDT8_NV, DSDT_MAG_NV,
DSDT8_MAG8_NV, DSDT_MAG_INTENSITY_NV, or DSDT8_MAG8_INTENSITY8_NV.

INVALID_OPERATION is generated when TexEnv is called and the
SHADER_OPERATION_NV parameter for texture unit 0 is assigned
one of OFFSET_TEXTURE_2D_NV, OFFSET_TEXTURE_2D_SCALE_NV,
OFFSET_TEXTURE_RECTANGLE_NV, OFFSET_TEXTURE_RECTANGLE_SCALE_NV,
DEPENDENT_AR_TEXTURE_2D_NV, DEPENDENT_GB_TEXTURE_2D_NV,
DOT_PRODUCT_NV, DOT_PRODUCT_DEPTH_REPLACE_NV,
DOT_PRODUCT_TEXTURE_2D_NV, DOT_PRODUCT_TEXTURE_RECTANGLE_NV,
DOT_PRODUCT_TEXTURE_3D_NV, DOT_PRODUCT_TEXTURE_CUBE_MAP_NV,
DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV, DOT_PRODUCT_REFLECT_CUBE_MAP_NV.
DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV,
OFFSET_PROJECTIVE_TEXTURE_2D_NV,
OFFSET_PROJECTIVE_TEXTURE_2D_SCALE_NV,
OFFSET_PROJECTIVE_TEXTURE_RECTANGLE_NV,
OFFSET_PROJECTIVE_TEXTURE_RECTANGLE_SCALE_NV,
OFFSET_HILO_TEXTURE_2D_NV, OFFSET_HILO_TEXTURE_RECTANGLE_NV,
OFFSET_HILO_PROJECTIVE_TEXTURE_2D_NV,
OFFSET_HILO_PROJECTIVE_TEXTURE_RECTANGLE_NV,
DEPENDENT_HILO_TEXTURE_2D_NV, DEPENDENT_RGB_TEXTURE_3D_NV,
DEPENDENT_RGB_TEXTURE_CUBE_MAP_NV, DOT_PRODUCT_PASS_THROUGH_NV,
DOT_PRODUCT_TEXTURE_1D_NV, or DOT_PRODUCT_AFFINE_DEPTH_REPLACE_NV."
```

New State

UPDATE lines in Table 6.TextureShaders.

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
SHADER_OPERATION_NV	TxZ37	GetTexEnviv	NONE	Texture shader operation	3.8.13	texture
RGBA_UNSIGNED_- DOT_PRODUCT_MAPPING_NV	TxZ3	GetTexEnviv	UNSIGNED_IDENTITY_NV	Texture shader RGBA dot product mapping	3.8.13	texture

* SHADER_OPERATION_NV: Z21 in NV_texture_shader (and Z23 in
  NV_texture_shader2) is now Z37 with NV_texture_shader3.

* RGBA_UNSIGNED_DOT_PRODUCT_MAPPING_NV: Z2 in NV_texture_shader is now
  Z3 with NV_texture_shader3.

[The "Tx" type prefix means that the state is per-texture unit.]

[The "Zn" type is an n-valued integer where n is the
  implementation-dependent number of texture units supported.]

New Implementation State

None

Revision History

November 15, 2001 - document that depth replace is after polygon offset.

June 5, 2002 - Driver implementations before this date incorrectly swap the HI and LO components of GL_HILO8_NV and GL_SIGNED_HILO8_NV textures. Drivers after this date have fixed the problem and match the specified behavior.

March 5, 2007 - Corrected some enum names.

Name

    NV_transform_feedback

Name Strings

    GL_NV_transform_feedback

Contributors

    Cliff Woolley
    Nick Carter

Contact

    Barthold Lichtenbelt (blichtenbelt 'at' nvidia.com)
    Pat Brown (pbrown 'at' nvidia.com)
    Eric Werness (ewerness 'at' nvidia.com)

Status

    Shipping for GeForce 8 Series (November 2006)

Version

```
    Last Modified Date:         02/04/2008
    NVIDIA Revision:            14
```

Number

    341

Dependencies

    OpenGL 1.5 is required.

    This extension interacts with EXT_timer_query.

    NV_vertex_program4, NV_geometry_program4 and NV_gpu_program4 affect this
    extension.

    EXT_geometry_shader4 trivially interacts with this extension.

    This extension has an OpenGL Shading Language component. As such it
    interacts with ARB_shader_objects and OpenGL 2.0.

    This extension is written against the OpenGL 2.0 specification.

Overview

    This extension provides a new mode to the GL, called transform feedback,
    which records vertex attributes of the primitives processed by the GL.
    The selected attributes are written into buffer objects, and can be
    written with each attribute in a separate buffer object or with all
    attributes interleaved into a single buffer object. If a geometry program
    or shader is active, the primitives recorded are those emitted by the
    geometry program. Otherwise, transform feedback captures primitives whose

vertex are transformed by a vertex program or shader, or by fixed-function vertex processing. In either case, the primitives captured are those generated prior to clipping. Transform feedback mode is capable of capturing transformed vertex data generated by fixed-function vertex processing, outputs from assembly vertex or geometry programs, or varying variables emitted from GLSL vertex or geometry shaders.

The vertex data recorded in transform feedback mode is stored into buffer objects as an array of vertex attributes. The regular representation and the use of buffer objects allows the recorded data to be processed directly by the GL without requiring CPU intervention to copy data. In particular, transform feedback data can be used for vertex arrays (via vertex buffer objects), as the source for pixel data (via pixel buffer objects), as program constant data (via the NV_parameter_buffer_object or EXT_bindable_uniform extension), or via any other extension that makes use of buffer objects.

This extension introduces new query object support to allow transform feedback mode to operate asynchronously. Query objects allow applications to determine when transform feedback results are complete, as well as the number of primitives processed and written back to buffer objects while in transform feedback mode. This extension also provides a new rasterizer discard enable, which allows applications to use transform feedback to capture vertex attributes without rendering anything.

New Procedures and Functions

```
    void BindBufferRangeNV(enum target, uint index, uint buffer,
                           intptr offset, sizeiptr size)
    void BindBufferOffsetNV(enum target, uint index, uint buffer,
                            intptr offset)
    void BindBufferBaseNV(enum target, uint index, uint buffer)
    void TransformFeedbackAttribsNV(sizei count, const int *attribs,
                                    enum bufferMode)
    void TransformFeedbackVaryingsNV(uint program, sizei count,
                                     const int *locations,
                                     enum bufferMode)
    void BeginTransformFeedbackNV(enum primitiveMode)
    void EndTransformFeedbackNV()

    int GetVaryingLocationNV(uint program, const char *name)
    void GetActiveVaryingNV(uint program, uint index,
                            sizei bufSize, sizei *length, sizei *size,
                            enum *type, char *name)
    void ActiveVaryingNV(uint program, const char *name)
    void GetTransformFeedbackVaryingNV(uint program, uint index,
                                       int *location)

    void GetIntegerIndexedvEXT(enum param, uint index, int *values);
    void GetBooleanIndexedvEXT(enum param, uint index, boolean *values);
```

(Note: These indexed query functions are provided in the EXT_draw_buffers2 extension. The boolean query is not useful for any queryable value in this extension, but is supported for completeness and consistency with base GL typed "Get" functions.)

New Tokens

```
    Accepted by the <target> parameters of BindBuffer, BufferData,
    BufferSubData, MapBuffer, UnmapBuffer, GetBufferSubData,
    GetBufferPointerv, BindBufferRangeNV, BindBufferOffsetNV and
    BindBufferBaseNV:

      TRANSFORM_FEEDBACK_BUFFER_NV                      0x8C8E

    Accepted by the <param> parameter of GetIntegerIndexedvEXT and
    GetBooleanIndexedvEXT:

      TRANSFORM_FEEDBACK_BUFFER_START_NV                0x8C84
      TRANSFORM_FEEDBACK_BUFFER_SIZE_NV                 0x8C85
      TRANSFORM_FEEDBACK_RECORD_NV                      0x8C86

    Accepted by the <param> parameter of GetIntegerIndexedvEXT and
    GetBooleanIndexedvEXT, and by the <pname> parameter of GetBooleanv,
    GetDoublev, GetIntegerv, and GetFloatv:

      TRANSFORM_FEEDBACK_BUFFER_BINDING_NV              0x8C8F

    Accepted by the <bufferMode> parameter of TransformFeedbackAttribsNV and
    TransformFeedbackVaryingsNV:

      INTERLEAVED_ATTRIBS_NV                            0x8C8C
      SEPARATE_ATTRIBS_NV                               0x8C8D

    Accepted by the <target> parameter of BeginQuery, EndQuery, and
    GetQueryiv:

      PRIMITIVES_GENERATED_NV                           0x8C87
      TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN_NV          0x8C88

    Accepted by the <cap> parameter of Enable, Disable, and IsEnabled, and by
    the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and
    GetDoublev:

      RASTERIZER_DISCARD_NV                             0x8C89

    Accepted by the <pname> parameter of GetBooleanv, GetDoublev, GetIntegerv,
    and GetFloatv:

      MAX_TRANSFORM_FEEDBACK_INTERLEAVED_COMPONENTS_NV  0x8C8A
      MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS_NV        0x8C8B
      MAX_TRANSFORM_FEEDBACK_SEPARATE_COMPONENTS_NV     0x8C80
      TRANSFORM_FEEDBACK_ATTRIBS_NV                     0x8C7E

    Accepted by the <pname> parameter of GetProgramiv:

      ACTIVE_VARYINGS_NV                                0x8C81
      ACTIVE_VARYING_MAX_LENGTH_NV                      0x8C82
      TRANSFORM_FEEDBACK_VARYINGS_NV                    0x8C83
```

    Accepted by the <pname> parameter of GetBooleanv, GetDoublev, GetIntegerv,
    GetFloatv, and GetProgramiv:

TRANSFORM_FEEDBACK_BUFFER_MODE_NV	0x8C7F

    Accepted by the <attribs> parameter of TransformFeedbackAttribsNV:

BACK_PRIMARY_COLOR_NV	0x8C77
BACK_SECONDARY_COLOR_NV	0x8C78
TEXTURE_COORD_NV	0x8C79
CLIP_DISTANCE_NV	0x8C7A
VERTEX_ID_NV	0x8C7B
PRIMITIVE_ID_NV	0x8C7C
GENERIC_ATTRIB_NV	0x8C7D
POINT_SIZE	0x0B11
FOG_COORDINATE	0x8451
SECONDARY_COLOR_NV	0x852D
PRIMARY_COLOR	0x8577
POSITION	0x1203
LAYER_NV	0x8DAA

(note: POINT_SIZE, FOG_COORDINATE, PRIMARY_COLOR, and POSITION are defined in the core OpenGL specification; SECONDARY_COLOR_NV is defined in NV_register_combiners.)

Returned by the <type> parameter of GetActiveVaryingNV:

UNSIGNED_INT_VEC2_EXT	0x8DC6
UNSIGNED_INT_VEC3_EXT	0x8DC7
UNSIGNED_INT_VEC4_EXT	0x8DC8

(note: All three of these are defined in the EXT_gpu_shader4 extension.)

Additions to Chapter 2 of the OpenGL 2.0 Specification (OpenGL Operation)

Insert three new sections between Sections 2.11, Coordinate Transforms and 2.12, Clipping:

(Move the "Asynchronous Queries" language out of Section 4.1.7)

Section 2.X, Asynchronous Queries

Asynchronous queries provide a mechanism to return information about the processing of a sequence of GL commands. There are two query types supported by the GL. Transform feedback queries (section 2.Y) returns information on the number of vertices and primitives processed by the GL and written to one or more buffer objects. Occlusion queries (section 4.1.7.1) count the number of fragments or samples that pass the depth test.

The results of asynchronous queries are not returned by the GL immediately after the completion of the last command in the set; subsequent commands can be processed while the query results are not complete. When available, the query results are stored in an associated query object. The commands described in section 6.1.12 provide mechanisms to determine when query results are available and return the actual results of the query. The name space for query objects is the unsigned integers, with zero reserved by the GL.

Each type of query supported by the GL has an active query object name. If the active query object name for a query type is non-zero, the GL is currently tracking the information corresponding to that query type and the query results will be written into the corresponding query object. If the active query object for a query type name is zero, no such information is being tracked.

A query object is created by calling

```
  void BeginQuery(enum target, uint id);
```

with an unused name <id>. <target> indicates the type of query to be performed; valid values of <target> are defined in subsequent sections. When a query object is created, the name <id> is marked as used and associated with a new query object.

BeginQuery sets the active query object name for the query type given by <target> to <id>. If BeginQuery is called with an <id> of zero, if the active query object name for <target> is non-zero, or if <id> is the active query object name for any query type, the error INVALID OPERATION is generated.

The command

```
  void EndQuery(enum target);
```

marks the end of the sequence of commands to be tracked for the query type given by <target>. The active query object for <target> is updated to indicate that query results are not available, and the active query object name for <target> is reset to zero. When the commands issued prior to EndQuery have completed and a final query result is available, the query object, active when EndQuery is, called is updated by the GL. The query object is updated to indicate that the query results are available and to contain the query result. If the active query object name for <target> is zero when EndQuery is called, the error INVALID_OPERATION is generated.

The command

```
  void GenQueries(sizei n, uint *ids);
```

returns <n> previously unused query object names in <ids>. These names are marked as used, but no object is associated with them until the first time they are used by BeginQuery.

Query objects are deleted by calling

```
  void DeleteQueries(sizei n, const uint *ids);
```

<ids> contains <n> names of query objects to be deleted. After a query object is deleted, its name is again unused. Unused names in <ids> are silently ignored.

Calling either GenQueries or DeleteQueries while any query of any target is active causes an INVALID_OPERATION error to be generated.

Query objects contain two pieces of state: a single bit indicating whether a query result is available, and an integer containing the query result value. The number of bits used to represent the query result is implementation-dependent. In the initial state of a query object, the result is available and its value is zero.

The necessary state for each query type is an unsigned integer holding the active query object name (zero if no query object is active), and any state necessary to keep the current results of an asynchronous query in progress.

Section 2.Y, Transform Feedback

In 'transform feedback' mode the vertices of transformed primitives are written out to one or more buffer objects. The vertices are fed back after the geometry shader stage, if it exists, or otherwise after vertex processing right before clipping (section 2.12) but after color clamping. Optionally the transformed vertices can be discarded after being stored into one or more buffer objects, or they can be passed on down to the clipping stage for further processing.

Transform feedback is started and finished by calling

```
  void BeginTransformFeedbackNV(enum primitiveMode)
```

and

```
  void EndTransformFeedbackNV(),
```

respectively. Transform feedback is said to be active after a call to BeginTransformFeedbackNV and inactive after a call to EndTransformFeedbackNV. Transform feedback is initially inactive. Transform feedback is performed after color clamping, but immediately before clipping in the OpenGL pipeline. <primitiveMode> is one of TRIANGLES, LINES, or POINTS, and specifies the output type of primitives that will be recorded into the buffer objects bound for transform feedback (see below). <primitiveMode> places a restriction on the primitive types that may be rendered during an instance of transform feedback. See table X.1.

Transform Feedback primitiveMode	allowed render primitive modes
POINTS	POINTS
LINES	LINES, LINE_LOOP, and LINE_STRIP
TRIANGLES	TRIANGLES, TRIANGLE_STRIP, TRIANGLE_FAN, QUADS, QUAD_STRIP, and POLYGON

Table X.1 Legal combinations between the transform feedback primitive mode, as passed to BeginTransformFeedbackNV and the current primitive mode.

If a geometry program or geometry shader is active, the output primitive type of the currently active program is used as the render primitive in table X.1, otherwise the Begin mode is used.

Quads and polygons will be tessellated and recorded as triangles (the order of tessellation within a primitive is undefined); primitives specified in strips or fans will be assembled and recorded as individual primitives. Incomplete primitives are not recorded. Begin or any operation that implicitly calls Begin (such as DrawElements) will generate INVALID_OPERATION if the begin mode is not an allowed begin mode for the current transform feedback buffer state. If a geometry program or geometry shader is active, its output primtive mode is used for the error check instead of the begin mode.

It is an invalid operation error to call BeginTransformFeedbackNV, TransformFeedbackBufferNV, TransformFeedbackVaryingsNV, TransformFeedbackAttribsNV, or UseProgram or LinkProgram on the currently active program object while transform feedback is active. It is an invalid operation error to call EndTransformFeedbackNV while transform feedback is inactive.

Transform feedback can operate in either INTERLEAVED_ATTRIBS_NV or SEPARATE_ATTRIBS_NV mode. In the INTERLEAVED_ATTRIBS_NV mode, several vertex attributes can be written, interleaved, into a single buffer object. In the SEPARATE_ATTRIBS_NV mode, vertex attributes are recorded, non-interleaved, into several buffer objects simultaneously.

It is an INVALID_OPERATION error to call BeginTransformFeedbackNV if there is no buffer object bound to index 0 (see the description of the BindBuffer* commands below) in INTERLEAVED_ATTRIBS_NV mode. It is also an INVALID_OPERATION error to call BeginTransformFeedbackNV if the number of buffer objects bound in SEPARATE_ATTRIBS_NV mode is less than the number of buffer objects required, as given by the current transform feedback state. It is also an INVALID_OPERATION error to call BeginTransformFeedbackNV if no attributes are specified to be captured in either separate or interleaved mode.

Buffer objects are made to be targets of transform feedback by calling one of

```
  void BindBufferRangeNV(enum target, uint index, uint buffer,
                         intptr offset, sizeiptr size)
  void BindBufferOffsetNV(enum target, uint index, uint buffer,
                          intptr offset)
  void BindBufferBaseNV(enum target, uint index, uint buffer)
```

where <target> is set to TRANSFORM_FEEDBACK_BUFFER_NV. Any of the three BindBuffer* commands perform the equivalent of BindBuffer(target, buffer). <buffer> specifies which buffer object to bind to the target at index number <index>. <index> exists for use with the SEPARATE_ATTRIBS_NV mode and must be less than the value of MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS_NV. <offset> specifies a starting offset into the buffer object <buffer>. <size> specifies the number of elements that can be written during transform feedback mode. This is useful to prevent the GL from writing past a certain position in the buffer object. Both <offset> and <size> are in basic machine units. The error INVALID_VALUE is generated if the value of <size> is less than or equal to zero. The error INVALID_VALUE is generated if <offset> or <size> are not word-aligned. The error INVALID_OPERATION is generated when any of the BindBuffer* commands is called while transform feedback is active.

BindBufferBaseNV is equivalent to calling BindBufferOffsetNV with an <offset> of 0. BindBufferOffsetNV is the equivalent of calling BindBufferRangeNV with <size> = sizeof(buffer) - <offset> and rounding <size> down so that it is word-aligned.

If recording the vertices of a primitive to the buffer objects being used for transform feedback purposes would result in either exceeding the limits of any buffer object's size, or in exceeding the end position <offset> + <size> - 1, as set by BindbufferRangeNV, then no vertices of the primitive are recorded, and the counter corresponding to the asynchronous query target TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN_NV (see Section 2.Z) is not incremented.

Two methods exist to specify which transformed vertex attributes are streamed to one, or more, buffer objects in transform feedback mode. If an OpenGL Shading Language vertex and/or geometry shader is active, then the state set with the TransformFeedbackVaryingsNV() command determines which attributes to record. If neither a vertex nor geometry shader is active, the state set with the TransformFeedbackAttribsNV() command determines which attributes to record.

When a program object containing a vertex shader and/or geometry shader is active, the set of vertex attributes recorded in transform feedback mode is specified by

```
  void TransformFeedbackVaryingsNV(uint program, sizei count,
                                   const int *locations,
                                   enum bufferMode)
```

This command sets the transform feedback state for <program> and specifies which varying variables to record when transform feedback is active. The array <locations> contains <count> locations of active varying variables, as queried with GetActiveVaryingNV(), to stream to a buffer object. See section 2.15.3. <bufferMode> is one of INTERLEAVED_ATTRIBS_NV or SEPARATE_ATTRIBS_NV. The error INVALID_OPERATION is generated if any value in <locations> does not reference an active varying variable, or if any value in <locations> appears more than once in the array. The same error is generated if <program> has not been linked successfully. The program object's state value TRANSFORM_FEEDBACK_BUFFER_MODE_NV will be set to <bufferMode> and the program object's state value TRANSFORM_FEEDBACK_VARYINGS_NV set to <count>. These values can be queried with GetProgramiv (see section 6.1.14).

In the INTERLEAVED_ATTRIBS_NV mode, varying variables are written, interleaved, into one buffer object. This is the buffer object bound to index 0. Varying variables are written out to that buffer object in the order that they appear in the array <locations>. The error INVALID_OPERATION is generated if the total number of components of all varying variables specified in the array <locations> is greater than MAX_TRANSFORM_FEEDBACK_INTERLEAVED_COMPONENTS_NV.

In the SEPARATE_ATTRIBS_NV mode, varying variables are recorded, non-interleaved, into several buffer objects simultaneously. The first varying variable in the array <locations> is written to the buffer bound to index 0. The last varying variable is written to the buffer object bound to index <count> - 1. No more than MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS_NV buffer objects can be written to simultaneously. The error INVALID_VALUE is generated if <count> is greater than that limit. Furthermore, the number of components for each varying variable in the array <locations> cannot exceed MAX_TRANSFORM_FEEDBACK_SEPARATE_COMPONENTS_NV. The error INVALID_VALUE is generated if any varying variable in <locations> exceeds this limit.

It is not necessary to (re-)link <program> after calling TransformFeedbackVaryingsNV(). Changes to the transform feedback state will be picked up right away after calling TransformFeedbackVaryingsNV().

The value for any attribute specified to be streamed to a buffer object but not actually written by a vertex or geometry shader is undefined.

When neither a vertex nor geometry shader is active, the vertex attributes produced by fixed-function vertex processing or an assembly vertex or geometry program can be recorded in transform feedback mode. The set of attributes to record is specified by

```
  void TransformFeedbackAttribsNV(sizei count, const int *attribs,
                                  enum bufferMode)
```

This command specifies which attributes to record into one, or more, buffer objects. The value TRANSFORM_FEEDBACK_BUFFER_MODE_NV will be set to <bufferMode> and the value TRANSFORM_FEEDBACK_ATTRIBS_NV set to <count>. The array <attribs> contains an interleaved representation of the attributes desired to be fed back containing 3*count values. For attrib i, the value at 3*i+0 is the enum corresponding to the attrib, as given in table X.2. The value at 3*i+1 is the number of components of the provided attrib to be fed back and is between 1 and 4. The value at 3*i+2 is the index for attribute enumerants corresponding to more than one real attribute. For an attribute enumerant corresponding to only one attribute, the index is ignored. For an attribute enumerant corresponding to more than one attribute, the error INVALID_VALUE is generated if the index value is outside the allowable range for that attribute.

attrib	permitted sizes	index?	GPU_program_4 result name
POSITION	1,2,3,4	no	position
PRIMARY_COLOR	1,2,3,4	no	color.front.primary
SECONDARY_COLOR_NV	1,2,3,4	no	color.front.secondary
BACK_PRIMARY_COLOR_NV	1,2,3,4	no	color.back.primary
BACK_SECONDARY_COLOR_NV	1,2,3,4	no	color.back.secondary
FOG_COORDINATE	1	no	fogcoord
POINT_SIZE	1	no	pointsize
TEXTURE_COORD_NV	1,2,3,4	yes	texcoord[index]
CLIP_DISTANCE_NV	1	yes	clip[index]
VERTEX_ID_NV	1	no	vertexid
PRIMITIVE_ID_NV	1	no	primid
GENERIC_ATTRIB_NV	1,2,3,4	yes	attrib[index]
LAYER_NV	1	no	layer

Table X.2: Transform Feedback Attribute Specifiers.The 'attrib' column

specifies which attribute to record. The 'permitted sizes' column indicates how many components of the attribute can be recorded. The 'index' column indicates if the attribute is indexed. The 'gpu program 4' column shows which result variable of a vertex or geometry program corresponds to the attribute to record.

The TransformFeedbackAttribsNV() command sets transform feedback state which is used both when the GL is in fixed-function vertex processing mode, as well as when an assembly vertex and/or geometry program is active.

The parameter <bufferMode> has the same meaning as described for TransformFeedbackVaryingsNV(). Attributes are either written interleaved, or into separate buffer objects, in the same manner as described earlier for TransformFeedbackVaryingsNV().

In the INTERLEAVED_ATTRIBS_NV mode, the error INVALID_VALUE is generated if the sum of the values of elements 3*i+1 in the array <attribs> is greater than MAX_TRANSFORM_FEEDBACK_INTERLEAVED_COMPONENTS_NV.

In the SEPARATE_ATTRIBS_NV mode, no more than MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS_NV buffer objects can be written to simultaneously. The error INVALID_VALUE is generated if <count> is greater than that limit.

The error INVALID_OPERATION is generated if any attribute appears more than once in the array <attribs>.

The value for any attribute specified to be streamed to a buffer object but not actually written by a vertex or geometry program is undefined. The values of PRIMITIVE_ID_NV or LAYER_NV for a vertex is defined if and only if a geometry program is active and that program writes to the result variables "result.primid" or "result.layer", respectively. The value of VERTEX_ID_NV is only defined if and only if a vertex program is active, no geometry program is active, and the vertex program writes to the output attribute "result.id".

Section 2.Z, Primitive Queries

Primitive queries use query objects to track the number of primitives generated by the GL and to track the number of primitives written to transform feedback buffers.

When BeginQuery is called with a <target> of PRIMITIVES_GENERATED_NV, the primitives-generated count maintained by the GL is set to zero. When the generated primitive query is active, the primitives-generated count is incremented every time a primitive reaches the Discarding Rasterization stage (see Section 3.x) right before rasterization. This counter counts the number of primitives emitted by a geometry shader, if active, possibly further tessellated into separate primitives during the transform-feedback stage, if active.

When BeginQuery is called with a <target> of TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN_NV, the transform-feedback-primitives-written count maintained by the GL is set to zero. When the transform feedback primitive written query is active, the transform-feedback-primitives-written count is incremented every time a primitive is recorded into a buffer object. If transform feedback is not active, this counter is not incremented. If the primitive does not fit in the buffer object, the counter is not incremented.

These two queries can be used together to determine if all primitives have been written to the bound feedback buffers; if both queries are run simultaneously and the query results are equal, all primitives have been written to the buffer(s). If the number of primitives written is less than the number of primitives generated, the buffer is full.

Modify section 2.15.3 "Shader Variables", page 75.

Change the second sentence in the first paragraph on p. 84 as follows:

. . . or read by a fragment shader, will count against this limit. The transformed vertex position (gl_Position) does not count against this limit.

Add the following paragraphs on p.84:

A varying variable is considered active if it is determined by the linker that the varying will actually be used when the executable code in a program object is executed. The linker will make this determination regardless of the transform-feedback state set with the TransformFeedbackVaryingsNV() command. In cases where the linker cannot make a conclusive determination, the varying will be considered active. It is possible to override this determination and force the linker to consider a varying variable as active by calling ActiveVaryingNV(). This can be useful in transform feedback mode if there are varying variables to be recorded but not otherwise needed.

To find the location of an active varying variable, call

```
  int GetVaryingLocationNV(uint program, const char *name)
```

This command will return the location of varying variable <name>. <name> is a null-terminated string without whitespace. If <name> is not the name of an active varying variable in <program>, -1 is returned. Locations for both user-defined as well as built-in varying variables can be queried. If <program> has not been successfully linked, the error INVALID_OPERATION is generated. After a program is linked, the location will not change, unless the program is re- linked. A valid name cannot be any portion of a single vector or matrix, but can be a single element of an array or the whole array. Note that varying variables cannot be structures.

To determine the set of active varying variables used by a program object, and their data types, use the command:

```
  void GetActiveVaryingNV(uint program, uint index,
                          sizei bufSize, sizei *length, sizei *size,
                          enum *type, char *name);
```

This command provides information about the varying selected by <index>. An <index> of 0 selects the first active varying variable, and an <index> of ACTIVE_VARYINGS_NV-1 selects the last active varying variable. The value of ACTIVE_VARYINGS_NV can be queried with GetProgramiv (see section 6.1.14). If <index> is greater than or equal to

ACTIVE_VARYINGS_NV, the error INVALID_VALUE is generated. The parameter <program> is the name of a program object for which the command LinkProgram has been issued in the past. It is not necessary for <program> to have been linked successfully. The link could have failed because the number of active varying variables exceeded the limit.

The name of the selected varying is returned as a null-terminated string in <name>. The actual number of characters written into <name>, excluding the null terminator, is returned in <length>. If <length> is NULL, no length is returned. The maximum number of characters that may be written into <name>, including the null terminator, is specified by <bufSize>. The returned varying name can be the name of a user defined varying variable or the name of a built- in varying (which begin with the prefix "gl_", see the OpenGL Shading Language specification for a complete list). The length of the longest varying name in program is given by ACTIVE_VARYING_MAX_LENGTH_NV, which can be queried with GetProgramiv (see section 6.1.14).

For the selected varying variable, its type is returned into <type>. The size of the varying is returned into <size>. The value in <size> is in units of the type returned in <type>. The type returned can be any of FLOAT, FLOAT_VEC2, FLOAT_VEC3, FLOAT_VEC4, INT, INT_VEC2, INT_VEC3, INT_VEC4, UNSIGNED_INT, UNSIGNED_INT_VEC2_EXT, UNSIGNED_INT_VEC3_EXT, UNSIGNED_INT_VEC4_EXT, FLOAT_MAT2, FLOAT_MAT3, or FLOAT_MAT4. If an error occurred, the return parameters <length>, <size>, <type> and <name> will be unmodified. This command will return as much information about active varying variables as possible. If no information is available, <length> will be set to zero and <name> will be an empty string. This situation could arise if GetActiveVaryingNV is issued after a failed link.

To force the linker to mark a varying variable as active, call

```
  void ActiveVaryingNV(uint program, const char *name)
```

to specify that the varying variable <name> in <program> should be marked as active when the program is next linked. In particular, it does not modify the list of active varying variables in a program object that has already been linked. For any varying variable in <program> not passed to ActiveVaryingNV, the linker will determine their active status. <name> must be a null-terminated string without whitespace. A valid name cannot be an element of an array, or any portion of a single vector or matrix. ActiveVaryingNV may be issued before any shader objects are attached to <program>. Hence, <name> can contain any string, including a name that is never used as a varying variable in any shader object. Such names are ignored by the GL.

The application is advised to force any varying variable live that it needs for transform feedback purposes. The set of active varying variables are linker dependent.

Additions to Chapter 3 of the OpenGL 2.0 Specification (Rasterization)

(Add new section 3.X, Discarding Rasterization)

Primitives can be optionally discarded before rasterization by calling Enable and Disable with RASTERIZER_DISCARD_NV. When enabled, primitives are discared right before the rasterization stage, but after the optional transform feedback stage. When disabled, primitives are passed through to the rasterization stage to be processed normally. RASTERIZER_DISCARD_NV applies to the DrawPixels, CopyPixels, Bitmap, Clear and Accum commands as well.

Additions to Chapter 4 of the OpenGL 2.0 Specification (Per-Fragment Operations and the Frame Buffer)

(Replace section 4.1.7, "Occlusion Queries", p. 204, with the following)

Occlusion queries use query objects to track the number of fragments or samples that pass the depth test. An occlusion query can be started and finished by calling BeginQuery and EndQuery, respectively, with a <target> of SAMPLES_PASSED.

When an occlusion query starts, the samples-passed count maintained by the GL is set to zero. When an occlusion query is active, the samples-passed count is incremented for each fragment that passes the depth test. If the value of SAMPLE BUFFERS is 0, then the samples- passed count is incremented by 1 for each fragment. If the value of SAMPLE BUFFERS is 1, then the samples-passed count is incremented by the number of samples whose coverage bit is set. However, implementations, at their discretion, may instead increase the samples-passed count by the value of SAMPLES if any sample in the fragment is covered. When an occlusion query finishes and all fragments generated by the commands issued prior to EndQuery have been generated, the samples-passed count is written to the corresponding query object as the query result value, and the query result for that object is marked as available.

If the samples-passed count overflows, (i.e., exceeds the value 2^n - 1, where n is the number of bits in the samples-passed count), its value becomes undefined. It is recommended, but not required, that implementations handle this overflow case by saturating at 2^n - 1 and incrementing no further.

Additions to Chapter 5 of the OpenGL 2.0 Specification (Special Functions)

(Add to section 5.4, Display Lists p. 237)

On p. 241, add the following to the list of vertex buffer object commands not compiled into a display list: BindBufferRangeNV, BindBufferOffsetNV, BindBufferBaseNV, TransformFeedbackAttribsNV, TransformFeedbackVaryingsNV, and ActiveVaryingNV.

Additions to Chapter 6 of the OpenGL 2.0 Specification (State and State Requests)

Modify the second paragraph of section 6.1.1 (Simple Queries) p244 to read as follows:

...<data> is a pointer to a scalar or array of the indicated type in which to place the returned data. The commands

```
  void GetIntegerIndexedvEXT(enum param, uint index, int *values);
  void GetBooleanIndexedvEXT(enum param, uint index, boolean *values);
```

are used to query indexed state. <target> is the name of the indexed state and <index> is the index of the particular element being queried. <data> is a pointer to a scalar or array of the indicated type in which to place the returned data. In addition ...

(Replace Section 6.1.12, Occlusion Queries, p. 254)

Section 6.1.12, Asynchronous Queries

The command

```
  boolean IsQuery(uint id);
```

returns TRUE if <id> is the name of a query object. If <id> is zero, or if <id> is a non-zero value that is not the name of a query object, IsQuery returns FALSE.

Information about a query target can be queried with the command

```
  void GetQueryiv(enum target, enum pname, int *params);
```

<target> identifies the query target and can be SAMPLES_PASSED for occlusion queries or PRIMITIVES_GENERATED_NV and TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN_NV for primitive queries.

If <pname> is CURRENT_QUERY, the name of the currently active query for <target>, or zero if no query is active, will be placed in <params>.

If <pname> is QUERY_COUNTER_BITS, the implementation-dependent number of bits used to hold the query result for <target> will be placed in params. The number of query counter bits may be zero, in which case the counter contains no useful information.

For primitive queries (PRIMITIVES_GENERATED_NV and TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN_NV) if the number of bits is non-zero, the minimum number of bits allowed is 32.

For occlusion queries (SAMPLES_PASSED), if the number of bits is non-zero, the minimum number of bits allowed is a function of the implementation's maximum viewport dimensions (MAX_VIEWPORT_DIMS). The counter must be able to represent at least two overdraws for every pixel in the viewport. The formula to compute the allowable minimum value (where n is the minimum number of bits) is:

```
  n = min(32, ceil(log_2(maxViewportWidth *
                         maxViewportHeight * 2))).
```

The state of a query object can be queried with the commands

```
  void GetQueryObjectiv(uint id, enum pname, int *params);
  void GetQueryObjectuiv(uint id, enum pname, uint *params);
```

If <id> is not the name of a query object, or if the query object named by <id> is currently active, then an INVALID_OPERATION error is generated.

If <pname> is QUERY_RESULT, then the query object's result value is returned as a single integer in <params>. If the value is so large in magnitude that it cannot be represented with the requested type, then the nearest value representable using the requested type is returned. If the number of query counter bits for any <target> is zero, then the result is returned as a single integer with a value of 0.

There may be an indeterminate delay before the above query returns. If <pname> is QUERY_RESULT_AVAILABLE, FALSE is returned if such a delay would be required, TRUE is returned otherwise. It must always be true that if any query object returns a result available of TRUE, all queries of the same type issued prior to that query must also return TRUE.

Querying the state for any given query object forces the corresponding query to complete within a finite amount of time.

If multiple queries are issued using the same object name prior to calling GetQueryObject[u]iv, the result and availability information returned will always be from the last query issued. The results from any queries before the last one will be lost if they are not retrieved before starting a new query on the same <target> and <id>.

(Add to Section 6.1.13, Buffer Objects, p. 255)

Add the following paragraph to the bottom of this section, p. 256.

To query which buffer objects are the target(s) when transform feedback is active, call GetIntegerIndexedvEXT() with <param> set to TRANSFORM_FEEDBACK_BUFFER_BINDING_NV. <index> has to be in the range 0 to MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS_NV - 1, otherwise the error INVALID_VALUE is generated. The name of the buffer object bound to <index> is returned in <values>. If no buffer object is bound for <index>, zero is returned in <values>.

To query the starting offset or size of the range of each buffer object binding used for transform feedback, call GetIntegerIndexedvEXT() with <param> set to TRANSFORM_FEEDBACK_BUFFER_START_NV or TRANSFORM_FEEDBACK_BUFFER_SIZE_NV respectively. The error INVALID_VALUE is generated if <index> not in the range 0 to MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS_NV - 1. If the parameter (starting offset or size) was not specified when the buffer object was bound, zero is returned. If no buffer object is bound to <index>, -1 is returned.

(Add a new Section 6.1.14 "Transform Feedback " and rename 6.1.14 to 6.1.15)

To query the attributes to stream to a buffer object when neither an OpenGL Shading Language vertex nor geometry shader is active, call GetIntegerIndexedvEXT() with <param> set to TRANSFORM_FEEDBACK_RECORD_NV. This will return three values in <values> for each <index>. The first value returned is the attribute. The second value the number of components of the attribute, and the third value the index of the attribute, if applicable. If the attribute is not indexed, the third component will return 0. The parameter <index> has to be in the range 0 to TRANSFORM_FEEDBACK_ATTRIBS_NV - 1, otherwise the error INVALID_VALUE is generated. If no data exists for <index> 0 is returned three times in <values>.

To query the attributes to stream to a buffer object when a vertex and/or geometry shader is active, use the command GetTransformFeedbackVaryingNV(), as explained in section 6.1.14.

(add to Section 6.1.14, Shader and Program Queries, p. 256)

Add the following paragraph to the bottom of page 257:

If <pname> is TRANSFORM_FEEDBACK_BUFFER_MODE_NV, the buffer mode, used when transform feedback is active, is returned. It can be one of SEPARATE_ATTRIBS_NV or INTERLEAVED_ATTRIBS_NV. If <pname> is TRANSFORM_FEEDBACK_VARINGS_NV, the number of varying variables to stream to one, or more, buffer objects are returned. If <pname> is ACTIVE_VARYINGS_NV, the number of active varying variables is returned. If no active varyings exist, 0 is returned. If <pname> is ACTIVE_VARYINGS_MAX_LENGTH_NV, the length of the longest active varying name, including a null terminator, is returned. If no active varying variable exists, 0 is returned.

The command

```
  void GetTransformFeedbackVaryingNV(uint program, uint index,
                                     int *location)
```

returns, for each <index>, the location of a varying variable to stream to a buffer object in <location>. The <index> element of the array <locations>, as passed to TransformFeedbackVaryingsNV, is returned. <index> has to be in the program object specific range 0 to TRANSFORM_FEEDBACK_VARYINGS_NV - 1, otherwise the error INVALID_VALUE is generated. If no location exists for <index>, -1 is returned. If <program> is not the name of a program object, or if program object has not been linked successfully, the error INVALID_OPERATION is generated.

Additions to Appendix A of the OpenGL 2.0 Specification (Invariance)

None.

Additions to the AGL/GLX/WGL Specifications

None.

Interactions with EXT_timer_query

EXT_timer_query is the first extension to generalize the BeginQuery and EndQuery mechanism introduced by ARB_occlusion_query and OpenGL 1.5 to cover an additional query type. This extension is the second. This extension is written against the OpenGL 2.0 specification and uses most of the modifications in the EXT_timer_query specification. If EXT_timer_query is supported, timer queries need to be added as a third query type.

Dependencies on NV_geometry_program4 and EXT_geometry_shader4

If NV_geometry_program4 is not supported, delete the reference to the output primitive type in Section 2.Y. Delete the reference to PRIMITIVE_ID_NV and LAYER_NV.

If EXT_geometry_shader4 is not supported, delete any reference to a geometry shader.

Dependencies on NV_vertex_program4 and NV_gpu_program4

If NV_vertex_program4 is not supported, delete any reference to VERTEX_ID_NV. If NV_gpu_program4 is not supported, table X.2 needs to refer to the "result" variables defined in the ARB_vertex_program specification instead.

Interactions with ARB_shader_objects and OpenGL 2.0

If neither ARB_shader_objects nor OpenGL 2.0 is supported, all references to shader and program objects, as well as varying variables, should be removed. This also means that functions including TransformFeedbackVaryingsNV, GetVaryingLocationNV, GetActiveVaryingNV, ActiveVaryingNV, and GetTransformFeedbackVaryingNV will not be supported, and enums that are relevant only in the context of shader and program objects will not be accepted.

Errors

The error INVALID_OPERATION is generated by BeginQuery if called with an <id> of zero, if the active query object name for <target> is non- zero, or if <id> is the active query object name for any query type.

The error INVALID_OPERATION is generated by EndQuery if the active query object name for <target> is zero.

The error INVALID_OPERATION is generated if Begin, or any command that performs an explicit Begin, is called when:

* A geometry program or shader is not active AND the begin mode does not match the allowed begin modes for the current transform feedback state as given by table X.1.

* A geometry program or shader is active AND the output primitive type (of the geometry program / shader) does not match the allowed begin modes for the current transform feedback state as given by table X.1.

The error INVALID_OPERATION is generated by BeginTransformFeedbackNV if there is no buffer object bound to index 0 in INTERLEAVED_ATTRIBS_NV mode.

The error INVALID_OPERATION is generated by BeginTransformFeedbackNV if the number of buffer objects bound in SEPARATE_ATTRIBS_NV mode is less than the number of buffer objects required, as given by the current transform feedback state.

The error INVALID_OPERATION is generated by BeginTransformFeedbackNV if no attributes are specified to be captured.

The error INVALID_OPERATION is generated by BeginTransformFeedbackNV, TransformFeedbackBufferNV, TransformFeedbackVaryingsNV, TransformFeedbackAttribsNV, or UseProgram or LinkProgram, called on the currently in use program object, while transform feedback is active.

The error INVALID_OPERATION is generated by EndTransformFeedbackNV while transform feedback is inactive.

The error INVALID_OPERATION is generated by BindBufferRangeNV, BindBufferOffsetNV or BindBufferBaseNV if <index> is greater or equal than MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS_NV.

The error INVALID_VALUE is generated by BindBufferRangeNV if the value of <size> <= 0.

The error INVALID_VALUE is generated by BindBufferRangeNV or BindBufferOffsetNV if <start> or <end> are not word-aligned.

The error INVALID_OPERATION is generated when any of the BindBuffer* commands is called while transform feedback is active.

The error INVALID_OPERATION is generated by TransformFeedbackVaryingsNV commands if any location appears more than once in the array <locations.

The error INVALID_OPERATION is generated by TransformFeedbackVaryingsNV if any location in <locations> references a non-existing varying variable.

The error INVALID_OPERATION is generated by TransformFeedbackVaryingsNV if <program> has not been linked successfully.

The error INVALID_OPERATION is generated by TransformFeedbackVaryingsNV in INTERLEAVED_ATTRIBS_NV mode if the total number of components of all varying variables specified in the array <locations> is greater than MAX_TRANSFORM_FEEDBACK_INTERLEAVED_COMPONENTS_NV.

The error INVALID_VALUE is generated by TransformFeedbackVaryingsNV or TransformFeedbackAttribsNV in SEPARATE_ATTRIBS_NV mode if <count> is greater than MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS_NV.

The error INVALID_VALUE is generated by TransformFeedbackVaryingsNV in SEPARATE_ATTRIBS_NV mode if the number of components for each varying variable in the array <locations> is greater than MAX_TRANSFORM_FEEDBACK_SEPARATE_COMPONENTS_NV.

The error INVALID_VALUE is generated by TransformFeedbackAttribsNV in INTERLEAVED_ATTRIBS_NV mode if the sum of the values of the components of the attributes in the array <attribs> is greater than MAX_TRANSFORM_FEEDBACK_INTERLEAVED_COMPONENTS_NV.

The error INVALID_OPERATION is generated by TransformFeedbackAttribsNV if an enum value is specified more than once in the array <attribs>.

The error INVALID_OPERATION is generated by TransformFeedbackAttribsNV if the number of components for each attribute in the array <attribs> is outside the range [0,4].

The error INVALID_VALUE is generated by TransformFeedbackAttribsNV if the index value is in the array <attribs> is outside the allowable range for an attribute enumerant corresponding to more than one real attribute.

The error INVALID_OPERATION is generated by GetVaryingLocationNV if <program> is not the name of a program object or if <program> has not been linked successfully.

The error INVALID_OPERATION is generated by GetActiveVaryingNV or ActiveVaryingNV if <program> is not the name of a program object.

The error INVALID_VALUE is generated by GetActiveVaryingNV if <index> is greater than or equal to ACTIVE_VARYINGS_NV.

The error INVALID_VALUE is generated by GetIntegerIndexedvEXT() or GetBooleanIndexedv() with <param> set to TRANSFORM_FEEDBACK_RECORD_NV if <index> is greater than or equal to TRANSFORM_FEEDBACK_ATTRIBS_NV.

The error INVALID_VALUE is generated by GetIntegerIndexedvEXT() or GetBooleanIndexedvEXT() with <param> set to TRANSFORM_FEEDBACK_BUFFER_BINDING_NV if <index> is greater than or equal to MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS_NV.

The error INVALID_VALUE is generated by GetTransformFeedbackVaryingsNV if <index> is greater than the program object specific value TRANSFORM_FEEDBACK_VARYINGS_NV - 1.

The error INVALID_OPERATION is generated by GetTransformFeedbackVaryingsNV if <program> is not the name of a program object, or if program object has not been linked successfully.

New State

(Add a new table: Table 6.X, Transform Feedback State)

Get Value	Type	Get Command	Init. Value	Description	Sec	Attrib
TRANSFORM_FEEDBACK_ BUFFER_MODE_NV	Z2	GetIntegerv	INTERLEAVED_ ATTRIBS_NV	Transform feedback mode	2.Y	-
TRANSFORM_FEEDBACK_ ATTRIBS_NV	Z2	GetIntegerv	0	Number of attributes to capture in transform feedback mode	2.Y	-
TRANSFORM_FEEDBACK_ BUFFER_BINDING_NV	Z+	GetIntegerv	0	Buffer object bound to generic bind point for transform feedback.	6.1.13	-
TRANSFORM_FEEDBACK_ RECORD_NV	nx3*Z+	GetInteger- IndexedvEXT	0	Name, component count, and index of each attribute captured	6.1.14	-
TRANSFORM_FEEDBACK_ BUFFER_BINDING_NV	nxZ+	GetInteger- IndexedvEXT	0	Buffer object bound to each transform feedback attribute stream.	6.1.13	-
TRANSFORM_FEEDBACK_ BUFFER_START_NV	nxZ+	GetInteger- IndexedvEXT	0	Start offset of binding range for each transform feedback attrib. stream	6.1.13	-
TRANSFORM_FEEDBACK_ BUFFER_SIZE_NV	nxZ+	GetInteger- IndexedvEXT	0	Size of binding range for each transform feedback attrib. stream	6.1.13	-

(Modify Table 6.37, p 298, updating the query object state to cover transform feedback.)

Get Value	Type	Get Command	Init. Value	Description	Sec	Attribute
CURRENT_QUERY	3xZ+	GetQueryiv	0	Active query object name (occlusion, timer, xform feedback)	2.X	-
QUERY_RESULT	3xZ+	GetQueryObjectiv	0	Query object result (samples passed, Time elapsed, feedback data amount)	2.X	-
QUERY_RESULT_AVAILABLE	3xZ+	GetQueryObjectiv	TRUE	Query object result available?	2.X	-

(Modify Table 6.29, p. 290, Program Object State. Add the following state.)

Get Value	Type	Get Command	Init. Value	Description	Sec	Attribute
ACTIVE_VARYINGS_NV	Z+	GetProgramiv	0	Number of active varyings	2.15.3	-
ACTIVE_VARYING_MAX_ LENGTH_NV	Z+	GetProgramiv	0	Maximum active varying name length	2.15.3	-
TRANSFORM_FEEDBACK_ BUFFER_MODE_NV	Z2	GetProgramiv	INTERLEAVED_ ATTRIBS_NV	Transform feedback mode for the program	6.1.14	-
TRANSFORM_FEEDBACK_ VARYINGS_NV	Z+	GetProgramiv	0	Number of varyings to stream to buffer object(s)	6.1.14	-
-	nxZ+	GetVarying- LocationNV	-	Location of each active varying variable	2.15.3	-
-	Z+	GetActive- VaryingNV	-	Size of each active varying variable	2.15.3	-
-	Z+	GetActive- VaryingNV	-	Type of each active varying variable	2.15.3	-
-	0+x- char	GetActive- VaryingNV	-	Name of each active varying variable	2.15.3	-
-	Z+	GetTransform- Feedback- VaryingNV	-	Varying location for one of the multiple varyings to capture	6.1.14	-

New Implementation Dependent State

(Modify Table 6.34, p. 295. Update the query object state to cover transform feedback.)

Get Value	Type	Get Command	Minimum Value	Description	Sec	Attribute
QUERY_COUNTER_BITS	2xZ+	GetQueryiv	see 6.1.12	Asynchronous query counter bits (occlusion, timer, tranform feedback queries)	6.1.12	-

(Add a new table, Table 6.X. Transform Feedback State.)

NOTE: In the "GetValue" columns below, MXFB stands for "MAX_TRANSFORM_FEEDBACK".

Get Value	Type	Get Command	Minimum Value	Description	Sec	Attribute
MXFB_INTERLEAVED_ COMPONENTS_NV	Z+	GetIntegerv	64	Max number of components to write to a single buffer in interleaved mode	2.Y	-
MXFB_SEPARATE_ ATTRIBS_NV	Z+	GetIntegerv	4	Max number of separate attributes or vayings that can be captured in transform feedback	2.Y	-
MXFB_SEPARATE COMPONENTS_NV	Z+	GetIntegerv	16	Max number of components per attribute or varying in separate mode	2.Y	-

Issues

1. *How does transform feedback differ from core GL feedback?*

   * Transform feedback writes vertex data to buffer objects, which allows the data returned to be used directly by vertex pulling. GL feedback mode writes vertex data to a buffer in system memory.

   * Transform feedback is done after transformation, but prior to clipping. The primitives returned contain the original transformed vertices produced by vertex or geometry program execution, and does not contain any primitives inserted by clipping.

   * Transform feedback supports only a single basic output primitive type (points, lines, or triangles), while core GL feedback mode supports all primitive types. Since only one primitive type is supported, the data returned does not contain tokens describing each primitive being fed back. Primitive tokens make the data returned by GL feedback mode irregular and unsuitable for vertex pulling.

2. *What should this extension be called?*

   RESOLVED: The current name is "NV_transform_feedback", playing off the fact that it is transformed primitives that are handled and the similarities to GL feedback mode.

3. *What happens if you bind a buffer for transform feedback that is currently bound for other purposes? Should we somehow detect this case and produce an error?*

   !!! NBC I feel strongly that we should follow the precedent for Map/Unmap. The reason that MapBuffer and UnmapBuffer are a precedent here is because while a buffer object is in the mapped state, no GL commands are allowed to operate on the buffer object's data. So by analogy, while a buffer is being used for transform feedback, no other GL commands should be allowed to operate on the buffer object's data. This includes initiating any rendering which would cause the GL to source data from an active transform feedback buffer object.

   UNRESOLVED

4. *Should this extension include any new buffer object binding targets, or should it overload ARRAY_BUFFER, or should we skip the binding target altogether in favor of a buffer object name accepted directly by the new GL commands?*

   RESOLVED: There are new binding points for XFB along with a new API (BindBufferBase etc) to set the internal binding points. A new binding point, TRANSFORM_FEEDBACK_BUFFER_NV is also introduced.

5. *Previous buffer object extensions provided a way to have existing GL commands reference a buffer object instead of a user-supplied buffer. Should the new commands introduced here allow referencing a user-supplied buffer in addition to a buffer object?*

   RESOLVED: No. A program can get the contents of the feedback buffer back to the CPU using MapBuffer and GetBufferSubData

6. *Is BeginTransformFeedback really necessary? Could the query just initiate the transform feedback mode?*

   RESOLUTION: Using BeginTransformFeedback and EndTransformFeedback gives a clean place to spec all of the transform-feedback-specific issues without cluttering up the query language. Also, the queries don't have to be done at the same time as beginning and ending the feedback process.

7. *What usage enums should be provided to glBufferData for use in conjunction with transform feedback?*

   RESOLVED: STREAM_COPY or STREAM_READ are expected to be the most common usages. If a buffer object is being written by the GL through transform feedback, and the contents of the buffer object are subsequently being consumed by the GL (e.g. by being used as a vertex buffer object), then this is a *_COPY usage. If the buffer object is being written by the GL through transform feedback, but is being consumed by the application (e.g. being mapped for read), this is a *_READ usage. The temporal (STREAM, STATIC, or DYNAMIC) component of the usage enum is determined by the ratio between how often the contents of the buffer object are modified and how often operations that source data from the buffer object occur.

8. *What should the behavior be when a buffer object is the active target of transform feedback, and it is deleted via DeleteBuffers?*

   RESOLVED: Deletion is deferred until the EndTransformFeedback if transform feedback is active.

9. *Should we allow more buffers to be bound than are used?*

   RESOLVED: Yes. The extra buffers are not in the way and can stay bound.

10. *Should we allow feedback to buffer lists with holes (i.e. 0 and 2 bound)?*

   RESOLVED: No. This makes for an ugly API with the potential for bugs, without any real benefit. The application can as well bind all buffers needed to incremented indices. It is an invalid operation to not have a buffer bound where one is required.

11. Why only one feedback primitive mode per feedback invocation?

RESOLVED: Having primitive tokens breaks up the stream and makes it less amenable to being read back in as a vertex buffer. Also, mixing multiple primitive types makes the counting of primitives less clear for the application.

12. Is RasterPos fed back?

RESOLVED: No.

13. Is DrawPixels/CopyPixels/Bitmap fed back?

RESOLVED: No. Rasterization occurs as normal, but there is no output to the feedback buffer. This is consistent with taking a tap out of the pipe before clipping.

14. Why do we need new BindBuffer functions?*

RESOLVED: All previous buffer object extensions have been retrofits of existing pointer-based APIs. New extensions built assuming buffer objects don't have that history, so need a new API. The functionality of these new functions combines the functionality of BindBuffer, to set the external bind point used by calls like MapBuffer and BufferSubData, with the functionality to set an internal bind point like VertexAttribPointer does.

15. How do the transform feedback indices, passed to the BindBuffer commands, work with multiple bindings?*

RESOLVED: The same way that they work with vertex arrays. There is one external bind point, TRANSFORM_FEEDBACK_BUFFER_NV. There are n internal bind points, selected with the <index> parameter to the BindBuffer* commands, where n is some implementation dependent limit. The BindBuffer* commands take the buffer passed and bind it to the external bind point, as well as to the selected internal bind point.

For example:

```
BindBufferOffsetNV(TRANSFORM_FEEDBACK_BUFFER_NV, 0, 1, 12);
// XFB index 0 points at buffer 1 with offset 12

BindBuffer(TRANSFORM_FEEDBACK_BUFFER_NV, 2);
// Buffer 2 is now bound to the external bind point. XFB index 0 still
// points at buffer 1

MapBuffer(TRANSFORM_FEEDBACK_BUFFER_NV, ...);
// Maps buffer 2
```

16. How are quads/quadstrips/polygons tesselated into triangles?

RESOLVED: In an implementation-dependent manner. OpenGL doesn't define quads or polygons in terms of triangles, so there is no one correct way to do it, and different gpus may implement the behavior differently. A quad may be split into two triangles in several different ways, and an application may not rely on this behavior.

17. How does this extension interact with display lists?

RESOLVED: Just like the VBO extension, none of the BindBuffer* commands are compiled into a display list.

18. Does polygon mode state affect the logic that determines if the transform feed back primitive mode and the render mode states are valid at the start of transform feedback mode?

RESOLVED: PolygonMode has no influence on the BeginTransFormFeedback primitiveMode check since it is performed later, in raster.

19. What to do with incomplete primitives?

RESOLVED: If there is no room to store one or more vertices of a primitive in a buffer object, none of the vertices in that primitive are written to the buffer. If a partial primitive enters transform feedback (i.e. only two vertices sent in triangles mode), none of the vertices in that primitive are written to the buffer object.

20. Why does TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN_NV have a TRANSFORM_FEEDBACK prefix but PRIMITIVES_GENERATED_NV doesn't?

RESOLVED: The number of primitives generated is independent of any feedback that is active. The number of primitives that are written is only valid for transform feedback - another extension could conceivably have a different way of writing out primitives that would require a similar but distinct token.

21. When a GLSL vertex shader is active, what happens in transform feedback mode if non-active varying variables are specified?

DISCUSSION: Active varying variables are varying variables, declared in the shader, that the linker determined are actually needed. As an optimization, the linker can discard the ones declared, but not needed. If non-active varying variables need to be fed into a buffer object, the linker should not perform this optimization.

There are three suggested resolutions to this problem:

1. The set of varying variables that need to be streamed to a buffer object in transform feedback mode are set as a property of the program object, and are taken into account during the link step. This means that changing the set means the application will have to re-link the program object in order to have the change take effect.

2. The set of varying variables that need to be streamed to a buffer object in transform feedback mode are specified after the program object has been linked. This is the most flexible option from the applications perspective, but this might mean that a) specifying this set could force the GL to re-link 'under the covers', and b) could mean that the GL runs out of varying variable slots because the combined total of the set of active varyings and the varyings to stream in transform feedback mode is too large.

3. This solution is a hybrid of the above two approaches. The set of potential varying variables that need to be streamed to a buffer object are set as a property of the program object. These varying variables are marked as active by the application and therefore cannot be eliminated during the link step. However, a sub-set of varying variables to actually stream to a buffer object can be changed without the application having to re-link the program object. This approach gives the application flexibility to change the set of varying variables to stream, while it eliminates the need for the GL to compile 'under the covers'.

RESOLUTION: Option 3 offers a good compromise, and therefore we'll go with that.

22. *Given option 3 in the previous resolution, how to specify that a varying variable has to be considered active by the linker?*

DISCUSSION: There are two approaches to the application specifying which varying variables are active. We can either provide a simple flag that specifies that all varying variables are considered active, or we can provide a more complex mechanism where the application can specify an individual varying variable as being active.

RESOLUTION: RESOLVED. The 'all or nothing' flag is a simple idea, but has a drawback when used with a 'uber-shader' that implements many paths to achieve an effect, but only one path is used during any run of the shader. In this case, a lot more varying variables might be flagged as active then really is necessary, running the risk of running out of resources. Therefore, we'll provide a mechanism for the application to specify on a per varying variable basis if it is active.

23. *Given the discussion in the previous issues, should a GetActiveVarying() command be added, modeled after the existing getActiveUniform() command?*

DISCUSSION: Such a command will return the list of active uniforms, after the program object has been linked. As per issue 22's resolution, the complete set of varying variables that could be streamed to a buffer object needs to be specified before the program object is linked.

It can be useful to an application to stream out a subset of the active varying variables or to find out the whole set of active varyings, especially since the set can be implementation dependent.

RESOLUTION: YES.

24. *What is proper use of the command ActiveVaryingNV()?*

RESOLVED: The application is well advised to force any varying variable live that it needs for transform feedback purposes. The set of active varying variables are linker dependent. For example, if a program object has no fragment shader, then the LinkProgram command cannot typically determine which built-in varying variables, output by a geometry or vertex shader, are active. This is because the fragment processing state can change, and therefore such a determination cannot be made until a render command is issued. Furthermore, any user-defined varyings are likely to be marked as non-active if there is no fragment shader because they are guaranteed to have no effect on fixed-function fragment processing. If there is both a vertex (or geometry) and fragment shader in a program object, the application can probably deduce what will be an active varying variable, or not. But beware of any (static) flow-control that the linker can use to do cross vertex- fragment optimization to cull any varying variables.

25. *Are primitives sent down the pipeline after transform feedback, or discarded?*

RESOLVED: Primitives can be optionally discarded before rasterization by calling Enable and Disable with RASTERIZER_DISCARD_NV. When enabled, primitives are discarded after vertex attributes are recorded into the buffer objects bound to transform feedback. When disabled, primitives are passed through to the rasterization stage to be clipped and rasterized normally. All rasterization operations are discarded, not just those that are fed back into the buffer.

This applies to DrawPixels, CopyPixels, Bitmap, Clear, Accum as well.

26. *If a varying is declared as an array, is the whole array streamed out?*

RESOLVED: No, the application has to specify which elements of an array it wants to stream out. Implementations might not be able to stream out a large number of components to a single buffer object. If that is the case, the application can stream each element of an array to a different buffer object in TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS mode.

27. *Is it possible to capture attributes when using the fixed-function pipeline?*

RESOLVED: Yes, there is nothing that precludes this. The application is responsible for sending down the needed vertex attributes and setting the GL state, as desired, for the attributes it wants to stream to a buffer object. Note that VERTEX_ID is not defined in fixed-function.

28. *Is it possible to record hardware-generated primitive ID values that would be available to a pixel shader?*

RESOLVED: Transform feedback can only record the primitive ID values emitted per-vertex by a geometry shader or program. While each primitive recorded for transform-feedback has a well-defined primitive ID, transform feedback is only capable of recording the attributes of individual vertices.

29. *Does transform feedback support the ability to capture per-vertex layer outputs, as provided by EXT_geometry_shader4 and NV_geometry_program4?*

RESOLVED: Yes. For GLSL shaders, it is sufficient to reference the built-in varying "gl_Layer". For assembly geometry programs, the original version of the spec did not provide an enum allowing you to name "result.layer" in TransformFeedbackAttribsNV. This was an oversight in the original spec, which was fixed by version 14. An updated driver will be required to take advantage of this capability; NVIDIA drivers supporting this extension published prior to February

    2008 will not be able to capture "result.layer". The value captured for
    LAYER_NV will be undefined unless a geometry program that writes
    "result.layer" is active.

Revision History

```
    Rev.    Date    Author    Changes
    ----  --------  --------  -----------------------------------------
     14   02/04/08  pbrown    Fixed a problem with the spec where we were
                              unable to record "result.layer" using the
                              assembly interface.  Added a new enum to
                              address.
```

Name

    NV_vertex_array_range

Name Strings

    GL_NV_vertex_array_range

Notice

IP Status

    NVIDIA Proprietary.

Status

    Shipping (version 1.1)

    Existing functionality is augmented by NV_vertex_array_range2.

Version

    NVIDIA Date: September 17, 2001 (version 1.1)

Number

    190

Dependencies

    None

Overview

The goal of this extension is to permit extremely high vertex processing rates via OpenGL vertex arrays even when the CPU lacks the necessary data movement bandwidth to keep up with the rate at which the vertex engine can consume vertices. CPUs can keep up if they can just pass vertex indices to the hardware and let the hardware "pull" the actual vertex data via Direct Memory Access (DMA). Unfortunately, the current OpenGL 1.1 vertex array functionality has semantic constraints that make such an approach hard. Hence, the vertex array range extension.

This extension provides a mechanism for deferring the pulling of vertex array elements to facilitate DMAed pulling of vertices for fast, efficient vertex array transfers. The OpenGL client need only pass vertex indices to the hardware which can DMA the actual index's vertex data directly out of the client address space.

The OpenGL 1.1 vertex array functionality specifies a fairly strict coherency model for when OpenGL extracts vertex data from a vertex array and when the application can update the in memory vertex array data. The OpenGL 1.1 specification says "Changes made to array data between the execution of Begin and the corresponding execution of End may affect calls to ArrayElement that are made within the same Begin/End period in non-sequential ways. That is, a call to ArrayElement that precedes a change to array data may access the changed data, and a call that follows a change to array data may access the original data."

This means that by the time End returns (and DrawArrays and DrawElements return since they have implicit Ends), the actual vertex array data must be transferred to OpenGL. This strict coherency model prevents us from simply passing vertex element indices to the hardware and having the hardware "pull" the vertex data out (which is often long after the End for the primitive has returned to the application).

Relaxing this coherency model and bounding the range from which vertex array data can be pulled is key to making OpenGL vertex array transfers faster and more efficient.

The first task of the vertex array range extension is to relax the coherency model so that hardware can indeed "pull" vertex data from the OpenGL client's address space long after the application has completed sending the geometry primitives requiring the vertex data.

The second problem with the OpenGL 1.1 vertex array functionality is the lack of any guidance from the API about what region of memory vertices can be pulled from. There is no size limit for OpenGL 1.1 vertex arrays. Any vertex index that points to valid data in all enabled arrays is fair game. This makes it hard for a vertex DMA engine to pull vertices since they can be potentially pulled from anywhere in the OpenGL client address space.

The vertex array range extension specifies a range of the OpenGL client's address space where vertices can be pulled. Vertex indices that access any array elements outside the vertex array range are specified to be undefined. This permits hardware to DMA from finite regions of OpenGL client address space, making DMA engine implementation tractable.

The extension is specified such that an (error free) OpenGL client using the vertex array range functionality could no-op its vertex array range commands and operate equivalently to using (if slower than) the vertex array range functionality.

Because different memory types (local graphics memory, AGP memory) have different DMA bandwidths and caching behavior, this extension includes a window system dependent memory allocator to allocate cleanly the most appropriate memory for constructing a vertex array range. The memory allocator provided allows the application to tradeoff the desired CPU read frequency, CPU write frequency, and memory priority while still leaving it up to OpenGL implementation the exact memory type to be allocated.

Issues

How does this extension interact with the compiled_vertex_array extension?

I think they should be independent and not interfere with each other. In practice, if you use NV_vertex_array_range, you can surpass the performance of compiled_vertex_array

Should some explanation be added about what happens when an OpenGL application updates its address space in regions overlapping with the currently configured vertex array range?

RESOLUTION: I think the right thing is to say that you get non-sequential results. In practice, you'll be using an old context DMA pointing to the old pages.

If the application change's its address space within the vertex array range, the application should call glVertexArrayRangeNV again. That will re-make a new vertex array range context DMA for the application's current address space.

If we are falling back to software transformation, do we still need to abide by leaving "undefined" vertices outside the vertex array range? For example, pointers that are not 32-bit aligned would likely cause a fall back.

RESOLUTION: No. The fact that vertex is "undefined" means we can do anything we want (as long as we send a vertex and do not crash) so it is perfectly fine for the software puller to grab vertex information not available to the hardware puller.

Should we give a programmer a sense of how big a vertex array range they can specify?

RESOLUTION: No. Just document it if there are limitations. Probably very hardware and operating system dependent.

Is it clear enough that language about ArrayElement also applies to DrawArrays and DrawElements?

Maybe not, but OpenGL 1.1 spec is clear that DrawArrays and DrawElements are defined in terms of ArrayElement.

Should glFlush be the same as glVertexArrayRangeFlush?

RESOLUTION: No. A glFlush is cheaper than a glVertexArrayRangeFlush though a glVertexArrayRangeFlushNV should do a flush.

If any the data for any enabled array for a given array element index falls outside of the vertex array range, what happens?

RESOLUTION: An undefined vertex is generated.

What error is generated in this case?

I don't know yet. We should make sure the hardware really does let us know when vertices are undefined.

Note that this is a little weird for OpenGL since most errors in OpenGL result in the command being ignored. Not in this case though.

Should this extension support an interface for allocating video and AGP memory?

RESOLUTION: YES. It seems like we should be able to leave the task of memory allocation to DirectDraw, but DirectDraw's asynchronous unmapping behavior and having to hold locks to update DirectDraw surfaces makes that mechanism to cumbersome.

Plus the API is a lot easier if we do it ourselves.

How do we decide what type of memory to allocate for the application?

RESOLUTION: Usage hints. The application rates the read frequency (how often will they read the memory), the write frequency (how often will they write the memory), and the priority (how important is this memory relative to other uses for the memory such as texturing) on a scale of 1.0 to 0.0. Using these hints and the size of the memory requsted, the OpenGL implementation decides where to allocate the memory.

We try to not directly expose particular types of memory (AGP, local memory, cached/uncached, etc) so future memory types can be supported by merely updating the OpenGL implementation.

Should the memory allocator functionality be available be a part of the GL or window system dependent (GLX or WGL) APIs?

RESOLUTION: The window system dependent API.

The memory allocator should be considered a window system/ operating system dependent operation. This also permits memory to be allocated when no OpenGL rendering contexts exist yet.

New Procedures and Functions

```
void VertexArrayRangeNV(sizei length, void *pointer)
void FlushVertexArrayRangeNV(void)
```

New Tokens

Accepted by the <cap> parameter of EnableClientState, DisableClientState, and IsEnabled:

```
    VERTEX_ARRAY_RANGE_NV              0x851D
```

Accepted by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

```
    VERTEX_ARRAY_RANGE_LENGTH_NV       0x851E
    VERTEX_ARRAY_RANGE_VALID_NV        0x851F
    MAX_VERTEX_ARRAY_RANGE_ELEMENT_NV  0x8520
```

Accepted by the <pname> parameter of GetPointerv:

```
    VERTEX_ARRAY_RANGE_POINTER_NV      0x8521
```

Additions to Chapter 2 of the OpenGL 1.1 Specification (OpenGL Operation)

After the discussion of vertex arrays (Section 2.8) add a description of the vertex array range:

"The command

```
   void VertexArrayRangeNV(sizei length, void *pointer)
```

specifies the current vertex array range. When the vertex array range is enabled and valid, vertex array vertex transfers from within the vertex array range are potentially faster. The vertex array range is a contiguous region of (virtual) address space for placing vertex arrays. The "pointer" parameter is a pointer to the base of the vertex array range. The "length" pointer is the length of the vertex array range in basic machine units (typically unsigned bytes).

The vertex array range address space region extends from "pointer" to "pointer + length - 1" inclusive. When specified and enabled, vertex array vertex transfers from within the vertex array range are potentially faster.

There is some system burden associated with establishing a vertex array range (typically, the memory range must be locked down). If either the vertex array range pointer or size is set to zero, the previously established vertex array range is released (typically, unlocking the memory).

The vertex array range may not be established for operating system dependent reasons, and therefore, not valid. Reasons that a vertex array range cannot be established include spanning different memory types, the memory could not be locked down, alignment restrictions are not met, etc.

The vertex array range is enabled or disabled by calling EnableClientState or DisableClientState with the symbolic constant VERTEX_ARRAY_RANGE_NV.

The vertex array range is either valid or invalid and this state can be determined by querying VERTEX_ARRAY_RANGE_VALID_NV. The vertex array range is valid when the following conditions are met:

- o VERTEX_ARRAY_RANGE_NV is enabled.

- o VERTEX_ARRAY is enabled.

- o VertexArrayRangeNV has been called with a non-null pointer and non-zero size.

- o The vertex array range has been established.

- o An implementation-dependent validity check based on the pointer alignment, size, and underlying memory type of the vertex array range region of memory.

- o An implementation-dependent validity check based on the current vertex array state including the strides, sizes, types, and pointer alignments (but not pointer value) for currently enabled vertex arrays.

- o Other implementation-dependent validaity checks based on other OpenGL rendering state.

Otherwise, the vertex array range is not valid. If the vertex array range is not valid, vertex array transfers will not be faster.

When the vertex array range is valid, ArrayElement commands may generate undefined vertices if and only if any indexed elements of the enabled arrays are not within the vertex array range or if the index is negative or greater or equal to the implementation-dependent value of MAX_VERTEX_ARRAY_RANGE_ELEMENT_NV. If an undefined vertex is generated, an INVALID_OPERATION error may or may not be generated.

The vertex array cohenecy model specifies when vertex data must be be extracted from the vertex array memory. When the vertex array range is not valid, (quoting the specification) `Changes made to array data between the execution of Begin and the corresponding execution of End may effect calls to ArrayElement that are made within the same Begin/End period in non-sequential ways. That is, a call to ArrayElement that precedes a change to array data may access the changed data, and a call that follows a change to array data may access the original data.'

When the vertex array range is valid, the vertex array coherency model is relaxed so that changes made to array data until the next "vertex array range flush" may affects calls to ArrayElement in non-sequential ways. That is a call to ArrayElement that precedes a change to array data (without an intervening "vertex array range flush") may access the changed data, and a call that follows a change (without an intervening "vertex array range flush") to array data may access original data.

A 'vertex array range flush' occurs when one of the following operations occur:

- o Finish returns.

- o FlushVertexArrayRangeNV returns.

- o VertexArrayRangeNV returns.

- o DisableClientState of VERTEX_ARRAY_RANGE_NV returns.

- o EnableClientState of VERTEX_ARRAY_RANGE_NV returns.

- o Another OpenGL context is made current.

    The client state required to implement the vertex array range
    consists of an enable bit, a memory pointer, an integer size,
    and a valid bit.

    If the memory mapping of pages within the vertex array range changes,
    using the vertex array range may or may not result in undefined data
    being fetched from the vertex arrays when the vertex array range is
    enabled and valid. To ensure that the vertex array range reflects
    the address space's current state, the application is responsible
    for calling VertexArrayRange again after any memory mapping changes
    within the vertex array range."llo

Additions to Chapter 5 of the OpenGL 1.1 Specification (Special Functions)

    Add to the end of Section 5.4 "Display Lists"

    "VertexArrayRangeNV and FlushVertexArrayRangeNV are not complied
    into display lists but are executed immediately.

    If a display list is compiled while VERTEX_ARRAY_RANGE_NV is
    enabled, the commands ArrayElement, DrawArrays, DrawElements,
    and DrawRangeElements are accumulated into a display list as
    if VERTEX_ARRAY_RANGE_NV is disabled."

Additions to the WGL interface:

    "When establishing a vertex array range, certain types of memory
    may be more efficient than other types of memory. The commands

```
       void *wglAllocateMemoryNV(sizei size,
                                 float readFrequency,
                                 float writeFrequency,
                                 float priority)
       void wglFreeMemoryNV(void *pointer)
```

    allocate and free memory that may be more suitable for establishing
    an efficient vertex array range than memory allocated by other means.
    The wglAllocateMemoryNV command allocates <size> bytes of contiguous
    memory.

    The <readFrequency>, <writeFrequency>, and <priority> parameters are
    usage hints that the OpenGL implementation can use to determine the
    best type of memory to allocate. These parameters range from 0.0
    to 1.0. A <readFrequency> of 1.0 indicates that the application
    intends to frequently read the allocated memory; a <readFrequency>
    of 0.0 indicates that the application will rarely or never read the
    memory. A <writeFrequency> of 1.0 indicates that the application
    intends to frequently write the allocated memory; a <writeFrequency>
    of 0.0 indicates that the application will rarely write the memory.
    A <priority> parameter of 1.0 indicates that memory type should be
    the most efficient available memory, even at the expense of (for
    example) available texture memory; a <priority> of 0.0 indicates that
    the vertex array range does not require an efficient memory type
    (for example, so that more efficient memory is available for other
    purposes such as texture memory).

The OpenGL implementation is free to use the <size>, <readFrequency>, <writeFrequency>, and <priority> parameters to determine what memory type should be allocated. The memory types available and how the memory type is determined is implementation dependent (and the implementation is free to ignore any or all of the above parameters).

Possible memory types that could be allocated are uncached memory, write-combined memory, graphics hardware memory, etc. The intent of the wglAllocateMemoryNV command is to permit the allocation of memory for efficient vertex array range usage. However, there is no requirement that memory allocated by wglAllocateMemoryNV must be used to allocate memory for vertex array ranges.

If the memory cannot be allocated, a NULL pointer is returned (and no OpenGL error is generated). An implementation that does not support this extension's memory allocation interface is free to never allocate memory (always return NULL).

The wglFreeMemoryNV command frees memory allocated with wglAllocateMemoryNV. The <pointer> should be a pointer returned by wglAllocateMemoryNV and not previously freed. If a pointer is passed to wglFreeMemoryNV that was not allocated via wglAllocateMemoryNV or was previously freed (without being reallocated), the free is ignored with no error reported.

The memory allocated by wglAllocateMemoryNV should be available to all other threads in the address space where the memory is allocated (the memory is not private to a single thread). Any thread in the address space (not simply the thread that allocated the memory) may use wglFreeMemoryNV to free memory allocated by itself or any other thread.

Because wglAllocateMemoryNV and wglFreeMemoryNV are not OpenGL rendering commands, these commands do not require a current context. They operate normally even if called within a Begin/End or while compiling a display list."

Additions to the GLX Specification

Same language as the "Additions to the WGL Specification" section except all references to wglAllocateMemoryNV and wglFreeMemoryNV should be replaced with glXAllocateMemoryNV and glXFreeMemoryNV respectively.

Additional language:

"OpenGL implementations using GLX indirect rendering should fail to set up the vertex array range (failing to set the vertex array valid bit so the vertex array range functionality is not usable). Additionally, glXAllocateMemoryNV always fails to allocate memory (returns NULL) when used with an indirect rendering context."

GLX Protocol

None

Errors

INVALID_OPERATION is generated if VertexArrayRange or FlushVertexArrayRange is called between the execution of Begin and the corresponding execution of End.

INVALID_OPERATION may be generated if an undefined vertex is generated.

New State

Get Value	Get Command	Type	Initial Value	Attrib
VERTEX_ARRAY_RANGE_NV	IsEnabled	B	False	vertex-array
VERTEX_ARRAY_RANGE_POINTER_NV	GetPointerv	Z+	0	vertex-array
VERTEX_ARRAY_RANGE_LENGTH_NV	GetIntegerv	Z+	0	vertex-array
VERTEX_ARRAY_RANGE_VALID_NV	GetBooleanv	B	False	vertex-array

New Implementation Dependent State

Get Value	Get Command	Type	Minimum Value
MAX_VERTEX_ARRAY_RANGE_ELEMENT_NV	GetIntegerv	Z+	65535

NV10 Implementation Details

This section describes implementation-defined limits for NV10:

The value of MAX_VERTEX_ARRAY_RANGE_ELEMENT_NV is 65535.

This section describes bugs in the NV10 vertex array range. These bugs will be fixed in a future hardware release:

If VERTEX_ARRAY is enabled with a format of GL_SHORT and the vertex array range is valid, a vertex array vertex with an X, Y, Z, or W coordinate of -32768 is wrongly interpreted as zero. Example: the X,Y coordinate (-32768,-32768) is incorrectly read as (0,0) from the vertex array.

If TEXTURE_COORD_ARRAY is enabled with a format of GL_SHORT and the vertex array range is valid, a vertex array texture coord with an S, T, R, or Q coordinate of -32768 is wrongly interpreted as zero. Example: the S,T coordinate (-32768,-32768) is incorrectly read as (0,0) from the texture coord array.

This section describes the implementation-dependent validity checks for NV10.

o For the NV10 implementation-dependent validity check for the vertex array range region of memory to be true, all of the following must be true:

1. The <pointer> must be 32-byte aligned.

```
          2.  The underlying memory types must all be the same (all
              standard system memory -OR- all AGP memory -OR- all video
              memory).

     o  For the NV10 implementation-dependent validity check for the
        vertex array state to be true, all of the following must be
        true:

       1.  ( VERTEX_ARRAY must be enabled -AND-
             The vertex array stride must be less than 256 -AND-
             ( ( The vertex array type must be FLOAT -AND-
                 The vertex array stride must be a multiple of 4 bytes -AND-
                 The vertex array pointer must be 4-byte aligned -AND-
                 The vertex array size must be 2, 3, or 4 ) -OR-
               ( The vertex array type must be SHORT -AND-
                 The vertex array stride must be a multiple of 4 bytes -AND-
                 The vertex array pointer must be 4-byte aligned. -AND-
                 The vertex array size must be 2 ) -OR-
               ( The vertex array type must be SHORT -AND-
                 The vertex array stride must be a multiple of 8 bytes -AND-
                 The vertex array pointer must be 8-byte aligned. -AND-
                 The vertex array size must be 3 or 4 ) ) )

       2.  ( NORMAL_ARRAY must be disabled. ) -OR -
           ( NORMAL_ARRAY must be enabled -AND-
             The normal array size must be 3 -AND-
             The normal array stride must be less than 256 -AND-
             ( ( The normal array type must be FLOAT -AND-
                 The normal array stride must be a multiple of 4 bytes -AND-
                 The normal array pointer must be 4-byte aligned. ) -OR-
               ( The normal array type must be SHORT -AND-
                 The normal array stride must be a multiple of 8 bytes -AND-
                 The normal array pointer must be 8-byte aligned. ) ) )

       3.  ( COLOR_ARRAY must be disabled. ) -OR -
           ( COLOR_ARRAY must be enabled -AND-
             The color array type must be FLOAT or UNSIGNED_BYTE -AND-
             The color array stride must be a multiple of 4 bytes -AND-
             The color array stride must be less than 256 -AND-
             The color array pointer must be 4-byte aligned -AND-
             The color array size must be 3 or 4 )

       4.  ( SECONDARY_COLOR_ARRAY must be disabled. ) -OR -
           ( SECONDARY_COLOR_ARRAY must be enabled -AND-
             The secondary color array type must be FLOAT or UNSIGNED_BYTE -AND-
             The secondary color array stride must be a multiple of 4 bytes -AND-
             The secondary color array stride must be less than 256 -AND-
             The secondary color array pointer must be 4-byte aligned -AND-
             The secondary color array size must be 3 or 4 )

       5.  For texture units zero and one:

           ( TEXTURE_COORD_ARRAY must be disabled. ) -OR -
           ( TEXTURE_COORD_ARRAY must be enabled -AND-
             The texture coord array stride must be less than 256 -AND-
             ( ( The texture coord array type must be FLOAT -AND-
                 The texture coord array pointer must be 4-byte aligned. )
                 The texture coord array stride must be a multiple of 4 bytes -AND-
                 The texture coord array size must be 1, 2, 3, or 4 ) -OR-
               ( The texture coord array type must be SHORT -AND-
                 The texture coord array pointer must be 4-byte aligned. )
                 The texture coord array stride must be a multiple of 4 bytes -AND-
```

```
                The texture coord array size must be 1 ) -OR-
              ( The texture coord array type must be SHORT -AND-
                The texture coord array pointer must be 4-byte aligned. )
                The texture coord array stride must be a multiple of 4 bytes -AND-
                The texture coord array size must be 2 ) -OR-
              ( The texture coord array type must be SHORT -AND-
                The texture coord array pointer must be 8-byte aligned. )
                The texture coord array stride must be a multiple of 8 bytes -AND-
                The texture coord array size must be 3 ) -OR-
              ( The texture coord array type must be SHORT -AND-
                The texture coord array pointer must be 8-byte aligned. )
                The texture coord array stride must be a multiple of 8 bytes -AND-
                The texture coord array size must be 4 ) ) )

        6.  ( EDGE_FLAG_ARRAY must be disabled. )

        7.  ( VERTEX_WEIGHT_ARRAY_NV must be disabled. ) -OR -
            ( VERTEX_WEIGHT_ARRAY_NV must be enabled. -AND -
              The vertex weight array type must be FLOAT -AND-
              The vertex weight array size must be 1 -AND-
              The vertex weight array stride must be a multiple of 4 bytes -AND-
              The vertex weight array stride must be less than 256 -AND-
              The vertex weight array pointer must be 4-byte aligned )

        8.  ( FOG_COORDINATE_ARRAY must be disabled. ) -OR -
            ( FOG_COORDINATE_ARRAY must be enabled -AND-
              The chip in use must be an NV11 or NV15, not NV10 -AND-
              The fog coordinate array type must be FLOAT -AND-
              The fog coordinate array size must be 1 -AND-
              The fog coordinate array stride must be a multiple of 4 bytes -AND-
              The fog coordinate array stride must be less than 256 -AND-
              The fog coordinate array pointer must be 4-byte aligned )
```

o For the NV10 the implementation-dependent validity check based on other OpenGL rendering state is FALSE if any of the following are true:

1. (COLOR_LOGIC_OP is enabled -AND-
   The logic op is not COPY), except in the case of Quadro2 (Quadro2 Pro, Quadro2 MXR) products.

2. (LIGHT_MODEL_TWO_SIDE is true.)

3. Either texture unit is enabled and active with a texture with a non-zero border.

4. VERTEX_PROGRAM_NV is enabled.

5. Several other obscure unspecified reasons.

NV20 Implementation Details

This section describes implementation-defined limits for NV20:

The value of MAX_VERTEX_ARRAY_RANGE_ELEMENT_NV is 1048575.

This section describes the implementation-dependent validity checks for NV20.

- o For the NV20 implementation-dependent validity check for the vertex array range region of memory to be true, all of the following must be true:

    1. The <pointer> must be 32-byte aligned.

    2. The underlying memory types must all be the same (all standard system memory -OR- all AGP memory -OR- all video memory).

- o To determine whether the NV20 implementation-dependent validity check for the vertex array state is true, the following algorithm is used:

  The currently enabled arrays and their pointers, strides, and types are first determined using the value of VERTEX_PROGRAM_NV. If VERTEX_PROGRAM_NV is disabled, the standard GL vertex arrays are used. If VERTEX_PROGRAM_NV is enabled, the vertex attribute arrays take precedence over the standard vertex arrays. The following table, taken from the NV_vertex_program specification, shows the aliasing between the standard and attribute arrays:

Vertex Attribute Register Number	Conventional Per-vertex Parameter	Conventional Per-vertex Parameter Command	Conventional Component Mapping
0	vertex position	Vertex	x,y,z,w
1	vertex weights	VertexWeightEXT	w,0,0,1
2	normal	Normal	x,y,z,1
3	primary color	Color	r,g,b,a
4	secondary color	SecondaryColorEXT	r,g,b,1
5	fog coordinate	FogCoordEXT	fc,0,0,1
6	-	-	-
7	-	-	-
8	texture coord 0	MultiTexCoord(GL_TEXTURE0_ARB, ...)	s,t,r,q
9	texture coord 1	MultiTexCoord(GL_TEXTURE1_ARB, ...)	s,t,r,q
10	texture coord 2	MultiTexCoord(GL_TEXTURE2_ARB, ...)	s,t,r,q
11	texture coord 3	MultiTexCoord(GL_TEXTURE3_ARB, ...)	s,t,r,q
12	texture coord 4	MultiTexCoord(GL_TEXTURE4_ARB, ...)	s,t,r,q
13	texture coord 5	MultiTexCoord(GL_TEXTURE5_ARB, ...)	s,t,r,q
14	texture coord 6	MultiTexCoord(GL_TEXTURE6_ARB, ...)	s,t,r,q
15	texture coord 7	MultiTexCoord(GL_TEXTURE7_ARB, ...)	s,t,r,q

  For the validity check to be TRUE, the following must all be true:

    1. Vertex attribute 0's array must be enabled.
    2. EDGE_FLAG_ARRAY must be disabled.
    3. For all enabled arrays, all of the following must be true:
        - the stride must be less than 256
        - the type must be FLOAT, SHORT, or UNSIGNED_BYTE

- o For the NV20 the implementation-dependent validity check based on other OpenGL rendering state is FALSE only for a few obscure and unspecified reasons.

Revision History

January 10, 2001 - Added NV20 implementation details. Made several corrections to the NV10 implementation details. Specifically, noted that on the NV11 and NV15 architectures, the fog coordinate array may be used, and updated the section on other state that may cause the vertex array range to be invalid. Only drivers built after this date will support fog coordinate arrays on NV11 and NV15. Also fixed a few typos in the spec.

September 17, 2001 - Modified NV20 implementation details to remove all the pointer and stride restrictions, none of which are actually required. Only drivers built after this date will support arbitrary pointer offsets and strides. Also removed NV10 rules on non-zero strides, which cannot be used in OpenGL anyhow, and fixed a few other typos.

Name

    NV_vertex_array_range2

Name Strings

    GL_NV_vertex_array_range2

Notice

IP Status

    NVIDIA Proprietary.

Status

    Complete

Version

    NVIDIA Date: April 13, 2001
    $Id: //sw/main/docs/OpenGL/specs/GL_NV_vertex_array_range2.txt#2 $

Number

    232

Dependencies

    Assumes support for the NV_vertex_array_range extension (version 1.1).

    Support for NV_fence is recommended but not required.

Overview

    Enabling and disabling the vertex array range is specified by the
    original NV_vertex_array_range extension specification to flush the
    vertex array range implicitly. In retrospect, this semantic is
    extremely misconceived and creates terrible performance problems
    for any application that wishes to mix conventional vertex arrays
    with vertex arrange range-enabled vertex arrays.

    This extension provides a new token for enabling/disabling the
    vertex array range that does NOT perform an implicit vertex array
    range flush when the enable/disable is performed.

Issues

Should this extension expose a new enable that enables/disables the vertex array range enable/disable semantic of performing an implicit 'vertex array range flush' when GL_VERTEX_ARRAY_RANGE_NV is enabled or disabled, OR should it add a new enable token that acts identically to GL_VERTEX_ARRAY_RANGE_NV without the implicit flush?

RESOLUTION: The second option. Enabling/disabling GL_VERTEX_ARRAY_RANGE_WITHOUT_FLUSH_NV acts identically to enabling/disabling GL_VERTEX_ARRAY_RANGE_NV, just without the implicit flush.

Should GL_VERTEX_ARRAY_RANGE_WITHOUT_FLUSH_NV work with glIsEnabled?

RESOLUTION: NO. There is still just a single state boolean to query.

New Procedures and Functions

None

New Tokens

Accepted by the <cap> parameter of EnableClientState, DisableClientState:

```
    VERTEX_ARRAY_RANGE_WITHOUT_FLUSH_NV  0x8533
```

Additions to Chapter 2 of the OpenGL 1.1 Specification (OpenGL Operation)

Within the discussion of vertex arrays (Section 2.8) amended by the NV_vertex_array_range extension specification, change the discussion of enabling the vertex array range to:

The vertex array range is enabled or disabled by calling EnableClientState or DisableClientState with the symbolic constant VERTEX_ARRAY_RANGE_NV.

The vertex array range is also enabled or disabled by calling EnableClientState or DisableClientState with the symbolic constant VERTEX_ARRAY_RANGE_WITHOUT_FLUSH_NV. This second means to enable and disable the vertex array range does not perform an implicit vertex array range flush as described subsequently."

Within the discussion of vertex arrays (Section 2.8) amended by the NV_vertex_array_range extension specification, change the discussion of implicit vertex array range flushes to:

```
    "A 'vertex array range flush' occurs when one of the following
    operations occur:

       o  Finish returns.

       o  FlushVertexArrayRangeNV returns.

       o  VertexArrayRangeNV returns.

       o  DisableClientState of VERTEX_ARRAY_RANGE_NV returns.

       o  EnableClientState of VERTEX_ARRAY_RANGE_NV returns.

       o  Another OpenGL context is made current.

    However, use of VERTEX_ARRAY_RANGE_WITHOUT_FLUSH_NV with
    DisableClientState or EnableClientState does NOT perform an implicit
    vertex array range flush."
```

Additions to Chapter 5 of the OpenGL 1.1 Specification (Special Functions)

None

Additions to the WGL interface:

None

Additions to the GLX Specification

None

GLX Protocol

None

Errors

No new errors.

New State

None

New Implementation Dependent State

None

Revision History

4/13/2001 - token value for GL_VERTEX_ARRAY_RANGE_WITHOUT_FLUSH_NV
should be 0x8533 (was incorrectly typed as 0x8503)

Name

    NV_vertex_program

Name Strings

    GL_NV_vertex_program

Notice

IP Status

NVIDIA Proprietary.

Status

Version 1.9

Version

NVIDIA Date: February 24, 2004
$Id: //sw/main/docs/OpenGL/specs/GL_NV_vertex_program.txt#20 $

Number

233

Dependencies

Written based on the wording of the OpenGL 1.2.1 specification and requires OpenGL 1.2.1.

Requires support for the ARB_multitexture extension with at least two texture units.

EXT_point_parameters affects the definition of this extension.

EXT_secondary_color affects the definition of this extension.

EXT_fog_coord affects the definition of this extension.

EXT_vertex_weighting affects the definition of this extension.

ARB_imaging affects the definition of this extension.

Overview

Unextended OpenGL mandates a certain set of configurable per-vertex computations defining vertex transformation, texture coordinate generation and transformation, and lighting. Several extensions have added further per-vertex computations to OpenGL. For example, extensions have defined new texture coordinate generation modes (ARB_texture_cube_map, NV_texgen_reflection, NV_texgen_emboss), new vertex transformation modes (EXT_vertex_weighting), new lighting modes (OpenGL 1.2's separate specular and rescale normal functionality), several modes for fog distance generation (NV_fog_distance), and eye-distance point size attenuation (EXT_point_parameters).

Each such extension adds a small set of relatively inflexible per-vertex computations.

This inflexibility is in contrast to the typical flexibility provided by the underlying programmable floating point engines (whether micro-coded vertex engines, DSPs, or CPUs) that are traditionally used to implement OpenGL's per-vertex computations. The purpose of this extension is to expose to the OpenGL application writer a significant degree of per-vertex programmability for computing vertex parameters.

For the purposes of discussing this extension, a vertex program is a sequence of floating-point 4-component vector operations that determines how a set of program parameters (defined outside of OpenGL's begin/end pair) and an input set of per-vertex parameters are transformed to a set of per-vertex output parameters.

The per-vertex computations for standard OpenGL given a particular set of lighting and texture coordinate generation modes (along with any state for extensions defining per-vertex computations) is, in essence, a vertex program. However, the sequence of operations is defined implicitly by the current OpenGL state settings rather than defined explicitly as a sequence of instructions.

This extension provides an explicit mechanism for defining vertex program instruction sequences for application-defined vertex programs. In order to define such vertex programs, this extension defines a vertex programming model including a floating-point 4-component vector instruction set and a relatively large set of floating-point 4-component registers.

The extension's vertex programming model is designed for efficient hardware implementation and to support a wide variety of vertex programs. By design, the entire set of existing vertex programs defined by existing OpenGL per-vertex computation extensions can be implemented using the extension's vertex programming model.

Issues

What should this extension be called?

RESOLUTION: NV_vertex_program. DirectX 8 refers to its similar functionality as "vertex shaders". This is a confusing term because shaders are usually assumed to operate at the fragment or pixel level, not the vertex level.

Conceptually, what the extension defines is an application-defined program (admittedly limited by its sequential execution model) for processing vertices so the "vertex program" term is more accurate.

Additionally, some of the API machinery in this extension for describing programs could be useful for extending other OpenGL operations with programs (though other types of programs would likely look very different from vertex programs).

What terms are important to this specification?

```
  vertex program mode - when vertex program mode is enabled, vertices
  are transformed by an application-defined vertex program.

  conventional GL vertex transform mode - when vertex program mode
  is disabled (or the extension is not supported), vertices are
  transformed by GL's conventional texgen, lighting, and transform
  state.

  provoke - the verb that denotes the beginning of vertex
  transformation by either vertex program mode or conventional GL
  vertex transform mode.  Vertices are provoked when either glVertex
  or glVertexAttribNV(0, ...) is called.

  program target - a type or class of program.  This extension
  supports two program targets:  the vertex program and the vertex
  state program.  Future extensions could add other program targets.

  vertex program -  an application-defined vertex program used to
  transform vertices when vertex program mode is enabled.

  vertex state program - a program similar to a vertex program.
  Unlike a vertex program, a vertex state program runs outside of
  a glBegin/glEnd pair.  Vertex state programs do not transform
  a vertex.  They just update program parameters.

  vertex attribute - one of 16 4-component per-vertex parameters
  defined by this extension.  These attributes alias with the
  conventional per-vertex parameters.

  per-vertex parameter - a vertex attribute or a conventional
  per-vertex parameter such as set by glNormal3f or glColor3f.

  program parameter - one of 96 4-component registers available
  to vertex programs.  The state of these registers is shared
  among all vertex programs.
```

What part of OpenGL do vertex programs specifically bypass?

Vertex programs bypass the following OpenGL functionality:

- Normal transformation and normalization
- Color material
- Per-vertex lighting
- Texture coordinate generation
- The texture matrix
- The normalization of AUTO_NORMAL evaluated normals
- The modelview and projection matrix transforms
- The per-vertex processing in EXT_point_parameters
- The per-vertex processing in NV_fog_distance
- Raster position transformation
- Client-defined clip planes

Operations not subsumed by vertex programs

- The view frustum clip
- Perspective divide (division by w)
- The viewport transformation
- The depth range transformation
- Clamping the primary and secondary color to [0,1]
- Primitive assembly and subsequent operations
- Evaluator (except the AUTO_NORMAL normalization)

How specific should this specification be about precision?

RESOLUTION: Reasonable precision requirements are incorporated into the specification beyond the often vague requirements of the core OpenGL specification.

This extension essentially defines an instruction set and its corresponding execution environment. The instruction set specified may find applications beyond the traditional purposes of 3D vertex transformation, lighting, and texture coordinate generation that have fairly lax precision requirements. To facilitate such possibly unexpected applications of this functionality, minimum precision requirements are specified.

The minimum precision requirements in the specification are meant to serve as a baseline so that application developers can write vertex programs with minimal worries about precision issues.

What about when the "execution environment" involves support for other extensions?

This extension assumes support for functionality that includes a fog distance, secondary color, point parameters, and multiple texture coordinates.

There is a trade-off between requiring support for these extensions to guarantee a particular extended execution environment and requiring lots of functionality that everyone might not support.

Application developers will desire a high baseline of functionality so that OpenGL applications using vertex programs can work in the full context of OpenGL. But if too much is required, the implementation burden mandated by the extension may limit the number of available implementations.

Clearly we do not want to require support for 8 texture units even if the machinery is there for it. Still multitexture is a common and important feature for using vertex programs effectively. Requiring at least two texture units seems reasonable.

What do we say about the alpha component of the secondary color?

RESOLUTION: When vertex program mode is enabled, the alpha component of csec used for the color sum state is assumed always zero. Another downstream extension may actually make the alpha component written into the COL1 (or BFC1) vertex result register available.

Should client-defined clip planes operate when vertex program mode is enabled?

RESOLUTION. No.

OpenGL's client-defined clip planes are specified in eye-space. Vertex programs generate homogeneous clip space positions. Unlike the conventional OpenGL vertex transformation mode, vertex program mode requires no semantic equivalent to eye-space.

Applications that require client-defined clip planes can simulate OpenGL-style client-defined clip planes by generating texture coordinates and using alpha testing or other per-fragment tests such as NV_texture_shader's CULL_FRAGMENT_NV program to discard fragments. In many ways, these schemes provide a more flexible mechanism for clipping than client-defined clip planes.

Unfortunately, vertex programs used in conjunction with selection or feedback will not have a means to support client-defined clip planes because the per-fragment culling mechanisms described in the previous paragraph are not available in the selection or feedback render modes. Oh well.

Finally, as a practical concern, client-defined clip planes greatly complicate clipping for various hardware rasterization architectures.

How are edge flags handled?

RESOLUTION: Passed through without the ability to be modified by a vertex program. Applications are free to send edge flags when vertex program mode is enabled.

Should vertex attributes alias with conventional per-vertex parameters?

RESOLUTION. YES.

This aliasing should make it easy to use vertex programs with existing OpenGL code that transfers per-vertex parameters using conventional OpenGL per-vertex calls.

It also minimizes the number of per-vertex parameters that the hardware must maintain.

See Table X.2 for the aliasing of vertex attributes and conventional per-vertex parameters.

How should vertex attribute arrays interact with conventional vertex arrays?

RESOLUTION: When vertex program mode is enabled, a particular vertex attribute array will be used if enabled, but if disabled, and the corresponding aliased conventional vertex array is enabled (assuming that there is a corresponding aliased conventional vertex array for the particular vertex array), the conventional vertex array will be used.

This matches the way immediate mode per-vertex parameter aliasing works.

This does slightly complicate vertex array validation in program mode, but programmers using vertex arrays can simply enable vertex program mode without reconfiguring their conventional vertex arrays and get what they expect.

Note that this does create an asymmetry between immediate mode and vertex arrays depending on whether vertex program mode is enabled or not. The immediate mode vertex attribute commands operate unchanged whether vertex program mode is enabled or not. However the vertex attribute vertex arrays are used only when vertex program mode is enabled.

Supporting vertex attribute vertex arrays when vertex program mode is disabled would create a large implementation burden for existing OpenGL implementations that have heavily optimized conventional vertex arrays. For example, the normal array can be assumed to always contain 3 and only 3 components in conventional OpenGL vertex transform mode, but may contain 1, 2, 3, or 4 components in vertex program mode.

There is not any additional functionality gained by supporting vertex attribute arrays when vertex program mode is disabled, but there is lots of implementation overhead. In any case, it does not seem something worth encouraging so it is simply not supported. So vertex attribute arrays are IGNORED when vertex program mode is not enabled.

Ignoring VertexAttribute commands or treating VertexAttribute commands as an error when vertex program mode is enabled would likely add overhead for such a conditional check. The implementation overhead for supporting VertexAttribute commands when vertex program mode is disabled is not that significant. Additionally, it is likely that setting persistent vertex attribute state while vertex program mode is disabled may be useful to applications. So vertex attribute immediate mode commands are PERMITTED when vertex program mode is not enabled.

Colors and normals specified as ints, uints, shorts, ushorts, bytes, and ubytes are converted to floating-point ranges when supplied to core OpenGL as described in Table 2.6. Other per-vertex attributes such as texture coordinates and positions are not converted. How does this mix with vertex programs where all vertex attributes are supposedly treated identically?

RESOLUTION: Vertex attributes specified as bytes and ubytes are always converted as described in Table 2.6. All other formats are not converted according to Table 2.6 but simply converted directly to floating-point.

The ubyte type is converted because those types seem more useful for passing colors in the [0,1] range.

If an application desires a conversion, the conversion can be incorporated into the vertex program itself.

This also applies to vertex attribute arrays. However, by enabling a color or normal vertex array and not enabling the corresponding aliased vertex attribute array, programmers can get the conventional conversions for color and normal arrays (but only for the vertex attribute arrays that alias to the conventional color and normal arrays and only with the sizes/types supported by these color and normal arrays).

Should programs be C-style null-terminated strings?

RESOLUTION: No. Programs should be specified as an array of GLubyte with an explicit length parameter. OpenGL has no precedent for passing null-terminated strings into the API (though glGetString returns null-terminated strings). Null-terminated strings are problematic for some languages.

Should all existing OpenGL transform functionality and extensions be implementable as vertex programs?

RESOLUTION: Yes. Vertex programs should be a complete superset of what you can do with OpenGL 1.2 and existing vertex transform extensions.

To implement EXT_point_parameters, the GL_VERTEX_PROGRAM_POINT_SIZE_NV enable is introduced.

To implement two-sided lighting, the GL_VERTEX_PROGRAM_TWO_SIDE_NV enable is introduced.

How does glPointSize work with vertex programs?

RESOLUTION: If GL_VERTEX_PROGRAM_POINT_SIZE_NV is disabled, the size of points is determine by the glPointSize state. If enabled, the point size is determined per-vertex by the clamped value of the vertex result PSIZ register.

Can the currently bound vertex program id be deleted or reloaded?

RESOLUTION. Yes. When a vertex program id is deleted or reloaded when it is the currently bound vertex program, it is as if a rebind occurs after the deletion or reload.

In the case of a reload, the new vertex program will be used from then on. In the case of a deletion, the current vertex program will be treated as if it is nonexistent.

Should program objects have a mechanism for managing program residency?

RESOLUTION: Yes. Vertex program instruction memory is a limited hardware resource. glBindProgramNV will be faster if binding to a resident program. Applications are likely to want to quickly switch between a small collection of programs.

glAreProgramsResidentNV allows the residency status of a group of programs to be queried. This mimics glAreTexturesResident.

Instead of adopting the glPrioritizeTextures mechanism, a new glRequestResidentProgramsNV command is specified instead. Assigning priorities to textures has always been a problematic endeavor and few OpenGL implementations implemented it effectively. For the priority mechanism to work well, it requires the client to routinely update the priorities of textures.

The glRequestResidentProgramsNV indicates to the GL that a set of programs are intended for use together. Because all the programs are requesting residency as a group, drivers should be able to attempt to load all the requested programs at once (and remove from residency programs not in the group if necessary). Clients can use glAreProgramsResidentNV to query the relative success of the request.

glRequestResidentProgramsNV should be superior to loading programs on-demand because fragmentation can be avoided.

What happens when you execute a nonexistent or invalid program?

RESOLUTION: glBegin will fail with a GL_INVALID_OPERATION if the currently bound vertex program is nonexistent or invalid. The same applies to glRasterPos and any command that implies a glBegin.

Because the glVertex and glVertexAttribNV(0, ...) are ignored outside of a glBegin/glEnd pair (without generating an error) it is impossible to provoke a vertex program if the current vertex program is nonexistent or invalid. Other per-vertex parameters (for examples those set by glColor, glNormal, and glVertexAttribNV when the attribute number is not zero) are recorded since they are legal outside of a glBegin/glEnd.

For vertex state programs, the problem is simpler because glExecuteProgramNV can immediately fail with a GL_INVALID_OPERATION when the named vertex state program is nonexistent or invalid.

What happens when a matrix has been tracked into a set of program parameters, but then glTrackMatrixNV(GL_VERTEX_PROGRAM_NV, addr, GL_NONE, GL_IDENTITY_NV) is performed?

RESOLUTION: The specified program parameters stop tracking a matrix, but they retain the values of the matrix they were last tracking.

Can rows of tracked matrices be queried by querying the program parameters that track them?

RESOLUTION: Yes.

Discussing matrices is confusing because of row-major versus column-major issues. Can you give an example of how a matrix is tracked?

```
// When loaded, the first row is "1, 2, 3, 4", because of column-major
// (OpenGL spec) vs. row-major (C) differences.
GLfloat matrix[16] = { 1, 5, 9,  13,
                       2, 6, 10, 14,
                       3, 7, 11, 15,
                       4, 8, 12, 16 };
GLfloat row1[4], row2[4];

glMatrixMode(GL_MATRIX0_NV);
glLoadMatrixf(matrix);
glTrackMatrixNV(GL_VERTEX_PROGRAM_NV, 4, GL_MATRIX0_NV, GL_IDENTITY_NV);
glTrackMatrixNV(GL_VERTEX_PROGRAM_NV, 8, GL_MATRIX0_NV, GL_TRANSPOSE_NV);
glGetProgramParameterfvNV(GL_VERTEX_PROGRAM_NV, 5,
  GL_PROGRAM_PARAMETER_NV, row1);
/* row1 is now [ 5 6 7 8 ] */
glGetProgramParameterfvNV(GL_VERTEX_PROGRAM_NV, 9,
  GL_PROGRAM_PARAMETER_NV, row2);
/* row2 is now [ 2 6 10 14 ] because the tracked matrix is transposed */
```

Should evaluators be extended to evaluate arbitrary vertex attributes?

RESOLUTION: Yes. We'll support 32 new maps (16 for MAP1 and 16 for MAP2) that take priority over the conventional maps that they might alias to (only when vertex program mode is enabled).

These new maps always evaluate all four components. The rationale for this is that if we supported 1, 2, 3, or 4 components, that would add 128 (16*4*2) enumerants which is too many. In addition, if you wanted to evaluate two 2-component vertex attributes, you could instead generate one 4-component vertex attribute and use the vertex program with swizzling to treat this as two-components.

Moreover, we are assuming 4-component vector instructions so less than 4-component evaluations might not be any more efficient than 4-component evaluations. Implementations that use vector instructions such as Intel's SSE instructions will be easier to implement since they can focus on optimizing just the 4-component case.

How should GL_AUTO_NORMAL work with vertex programs?

RESOLUTION: GL_AUTO_NORMAL should NOT guarantee that the generated analytical normal be normalized. In vertex program mode, the current vertex program can easily normalize the normal if required.

This can lead to greater efficiency if the vertex program transforms the normal to another coordinate system such as eye-space with a transform that preserves vector length. Then a single normalize after transform is more efficient than normalizing after evaluation and also normalizing after transform.

Conceptually, the normalize mandated for AUTO_NORMAL in section 5.1 is just one of the many transformation operations subsumed by vertex programs.

Should the new vertex program related enables push/pop with GL_ENABLE_BIT?

RESOLUTION: Yes. Pushing and popping enable bits is easy. This includes the 32 new evaluator map enable bits. These evaluator enable bits are also pushed and popped using GL_EVAL_BIT.

Should all the vertex attribute state push/pop with GL_CURRENT_BIT?

RESOLUTION: Yes. The state is aliased with the conventional per-vertex parameter state so it really should push/pop.

Should all the vertex attrib vertex array state push/pop with GL_CLIENT_VERTEX_ARRAY_BIT?

RESOLUTION: Yes.

Should all the other vertex program-related state push/pop somehow?

RESOLUTION: No.

The other vertex program doesn't fit well with the existing bits. To be clear, GL_ALL_ATTRIB_BITS does not push/pop vertex program state other than enables.

Should we generate a GL_INVALID_OPERATION operation if updating a vertex attribute greater than 15?

RESOLUTION: Yes.

The other option would be to mask or modulo the vertex attribute index with 16. This is cheap, but it would make it difficult to increase the number of vertex attributes in the future.

If we check for the error, it should be a well predicted branch for immediate mode calls. For vertex arrays, the check is only required at vertex array specification time.

Hopefully this will encourage people to use vertex arrays over immediate mode.

Should writes to program parameter registers during a vertex program be supported?

RESOLUTION. No.

Writes to program parameter registers from within a vertex program would require the execution of vertex programs to be serialized with respect to each other. This would create an unwarranted implementation penalty for parallel vertex program execution implementations.

However vertex state programs may write to program parameter registers (that is the whole point of vertex state programs).

Should we support variously sized immediate mode byte and ubyte commands? How about for vertex arrays?

RESOLUTION. Only support the 4ub mode.

There are simply too many glVertexAttribNV routines. Passing less than 4 bytes at a time is inefficient. We expect the main use for bytes to be for colors where these will be unsigned bytes. So let's just support 4ub mode for bytes. This applies to vertex arrays too.

Should we support integer, unsigned integer, and unsigned short formats for vertex attributes?

RESOLUTION: No. It's just too many immediate mode entry points, most of which are not that useful. Signed shorts are supported however. We expect signed shorts to be useful for passing compact texture coordinates.

Should we support doubles for vertex attributes?

RESOLUTION: Yes. Some implementation of the extension might support double precision. Lots of math routines output double precision.

Should there be a way to determine where in a loaded program string the first parse error occurs?

RESOLUTION: Yes. You can query PROGRAM_ERROR_POSITION_NV.

Should program objects be shared among rendering contexts in the same manner as display lists and texture objects?

RESOLUTION: Yes.

How should this extension interact with color material?

RESOLUTION: It should not. Color material is a conventional OpenGL vertex transform mode. It does not have a place for vertex programs. If you want to emulate color material with vertex programs, you would simply write a program where the material parameters feed from the color vertex attribute.

Should there be a glMatrixMode or glActiveTextureARB style selector for vertex attributes?

RESOLUTION: No. While this would let us reduce a lot of enumerants down, it would make programming a hassle in lots of cases. Consider having to change the vertex attribute mode to enable a set of vertex arrays.

How should gets for vertex attribute array pointers?

RESOLUTION: Add new get commands. Using the existing calls would require adding 4 sets of 16 enumerants stride, type, size, and pointer. That's too many gets.

Instead add glGetVertexAttribNV and glGetVertexAttribPointervNV. glGetVertexAttribNV is also useful for querying the current vertex attribute.

glGet and glGetPointerv will not return vertex attribute array pointers.

Why is the address register numbered and why is it a vector register?

In the future, A0.y and A0.z and A0.w may exist. For this extension, only A0.x is useful. Also in the future, there may be more than one address register.

There's a nice consistency in thinking about all the registers as 4-component vectors even if the address register has only one usable component.

Should vertex programs and vertex state programs be required to have a header token and an end token?

RESOLUTION: Yes.

The "!!VP1.0" and "!!VSP1.0" tokens start vertex programs and vertex state programs respectively. Both types of programs must end with the "END" token.

The initial header token reminds the programmer what type of program they are writing. If vertex programs and vertex state programs are ever read from disk files, the header token can serve as a magic number for identifying vertex programs and vertex state programs.

The target type for vertex programs and vertex state programs can be distinguished based on their respective grammars independent of the initial header tokens, but the initial header tokens will make it easier for programmers to distinguish the two program target types.

We expect programs to often be generated by concatenation of program fragments. The "END" token will hopefully reduce bugs due to specifying an incorrectly concatenated program.

It's tempting to make these additional header and end tokens optional, but if there is a sanity check value in header and end tokens, that value is undermined if the tokens are optional.

What should be said about rendering invariances?

RESOLUTION: See the Appendix A additions below.

The justification for the two rules cited is to support multi-pass rendering when using vertex programs. Different rendering passes will likely use different programs so there must be some means of guaranteeing that two different programs can generate particular identical vertex results between different passes.

In practice, this does limit the type of vertex program implementations that are possible.

For example, consider a limited hardware implementation of vertex programs that uses a different floating-point implementation than the CPU's floating-point implementation. If the limited hardware implementation can only run small vertex programs (say the hardware provides on 4 temporary registers instead of the required 12), the implementation is incorrect and non-conformant if programs that only require 4 temporary registers use the vertex program hardware, but programs that require more than 4 temporary registers are implemented by the CPU.

This is a very important practical requirement. Consider a multi-pass rendering algorithm where one pass uses a vertex program that uses only 4 temporary registers, but a different pass uses a vertex program that uses 5 temporary registers. If two programs have instruction sequences that given the same input state compute identical resulting vertex positions, the multi-pass algorithm should generate identically positioned primitives for each pass.

But given the non-conformant vertex program implementation described above, this could not be guaranteed.

This does not mean that schemes for splitting vertex program implementations between dedicated hardware and CPUs are impossible. If the CPU and dedicated vertex program hardware used IDENTICAL floating-point implementations and therefore generated exactly identical results, the above described could work.

While these invariance rules are vital for vertex programs operating correctly for multi-pass algorithms, there is no requirement that conventional OpenGL vertex transform mode will be invariant with vertex program mode. A multi-pass algorithm should not assume that one pass using vertex program mode and another pass using conventional GL vertex transform mode will generate identically positioned primitives.

Consider that while the conventional OpenGL vertex program mode is repeatable with itself, the exact procedure used to transform vertices is not specified nor is the procedure's precision specified. The GL specification indicates that vertex coordinates are transformed by the modelview matrix and then transformed by the projection matrix. Some implementations may perform this sequence of transformations exactly, but other implementations may transform vertex coordinates by the composite of the modelview and projection matrices (one matrix transform instead of two matrix transforms in sequence). Given this implementation flexibility, there is no way for a vertex program author to exactly duplicate the precise computations used by the conventional OpenGL vertex transform mode.

The guidance to OpenGL application programs is clear. If you are going to implement multi-pass rendering algorithms that require certain invariances between the multiple passes, choose either vertex program mode or the conventional OpenGL vertex transform mode for your rendering passes, but do not mix the two modes.

What range of relative addressing offsets should be allowed?

RESOLUTION: -64 to 63.

Negative offsets are useful for accessing a table centered at zero without extra bias instructions. Having the offsets support much larger magnitudes just seems to increase the required instruction widths. The -64 to 63 range seems like a reasonable compromise.

When EXT_secondary_color is supported, how does the GL_COLOR_SUM_EXT enable affect vertex program mode?

RESOLUTION: The GL_COLOR_SUM_EXT enable has no affect when vertex program mode is enabled.

When vertex program mode is enabled, the color sum operation is always in operation. A program can "avoid" the color sum operation by not writing the COL1 (or BFC1 when GL_VERTEX_PROGRAM_TWO_SIDE_NV) vertex result registers because the default values of all vertex result registers is (0,0,0,1). For the color sum operation, the alpha value is always assumed zero. So by not writing the secondary color vertex result registers, the program assures that zero is added as part of the color sum operation.

If there is a cost to the color sum operation, OpenGL implementations may be smart enough to determine at program bind time whether a secondary color vertex result is generated and implicitly disable the color sum operation.

Why must RCP of 1.0 always be 1.0?

This is important for 3D graphics so that non-projective textures and orthogonal projections work as expected. Basically when q or w is 1.0, things should work as expected.

Stronger requirements such as "RCP of -1.0 must always be -1.0" are encouraged, but there is no compelling reason to state such requirements explicitly as is the case for "RCP of 1.0 must always be 1.0".

What happens when the source scalar value for the ARL instruction is an extremely positive or extremely negative floating-point value? Is there a problem mapping the value to a constrained integer range?

RESOLUTION: It is not a problem. Relative addressing can by offset by a limited range of offsets (-64 to 63). Relative addressing that falls outside of the 0 to 95 range of program parameter registers is automatically mapped to (0,0,0,0).

Clamping the source scalar value for ARL to the range -64 to 160 inclusive is sufficient to ensure that relative addressing is out of range.

How do you perform a 3-component normalize in three instructions?

```
#
# R1 = (nx,ny,nz)
#
# R0.xyz = normalize(R1)
# R0.w   = 1/sqrt(nx*nx + ny*ny + nz*nz)
#
DP3 R0.w, R1, R1;
RSQ R0.w, R0.w;
MUL R0.xyz, R1, R0.w;
```

How do you perform a 3-component cross product in two instructions?

```
#
# Cross product |  i     j     k   | into R2.
#               | R0.x  R0.y  R0.z |
#               | R1.x  R1.y  R1.z |
#
MUL R2, R0.zxyw, R1.yzxw;
MAD R2, R0.yzxw, R1.zxyw, -R2;
```

How do you perform a 4-component vector absolute value in one instruction?

```
  #
  # Absolute value is the maximum of the negative and positive
  # components of a vector.
  #
  # R1 = abs(R0)
  #
  MAX R1, R0, -R0;
```

How do you compute the determinant of a 3x3 matrix in three instructions?

```
  #
  # Determinant of | R0.x  R0.y  R0.z | into R3
  #                | R1.x  R1.y  R1.z |
  #                | R2.x  R2.y  R2.z |
  #
  MUL R3, R1.zxyw, R2.yzxw;
  MAD R3, R1.yzxw, R2.zxyw, -R3;
  DP3 R3, R0, R3;
```

How do you transform a vertex position by a 4x4 matrix and then perform a homogeneous divide?

```
  #
  # c[20] = modelview row 0
  # c[21] = modelview row 1
  # c[22] = modelview row 2
  # c[23] = modelview row 3
  #
  # result = R5
  #
  DP4 R5.w, v[OPOS], c[23];
  DP4 R5.x, v[OPOS], c[20];
  DP4 R5.y, v[OPOS], c[21];
  DP4 R5.z, v[OPOS], c[22];
  RCP R11, R5.w;
  MUL R5,R5,R11;
```

How do you perform a vector weighting of two vectors using a single weight?

```
  #
  # R2        = vector 0
  # R3        = vector 1
  # v[WGHT].x = scalar weight to blend vectors 0 and 1
  # result    = R2 * v[WGHT].x + R3 * (1-v[WGHT])
  #
  # this is because A*B + (1-A)*C = A*(B-C) + C
  #
  ADD R4, R2, -R3;
  MAD R4, v[WGHT].x, R4, R3;
```

*How do you reduce a value to some fundamental period such as 2*PI?*

```
  #
  # c[36] = (1.0/(2*PI), 2*PI, 0.0, 0.0)
  #
  # R1.x = input value
  # R2   = result
  #
  MUL R0, R1, c[36].x;
  EXP R4, R0.x;
  MUL R2, R4.y, c[36].y;
```

How do you implement a simple specular and diffuse lighting computation with an eye-space normal?

```
  !!VP1.0
  #
  # c[0-3]  = modelview projection (composite) matrix
  # c[4-7]  = modelview inverse transpose
  # c[32]   = normalized eye-space light direction (infinite light)
  # c[33]   = normalized constant eye-space half-angle vector (infinite viewer)
  # c[35].x = pre-multiplied monochromatic diffuse light color & diffuse material
  # c[35].y = pre-multiplied monochromatic ambient light color & diffuse material
  # c[36]   = specular color
  # c[38].x = specular power
  #
  # outputs homogenous position and color
  #
  DP4   o[HPOS].x, c[0], v[OPOS];
  DP4   o[HPOS].y, c[1], v[OPOS];
  DP4   o[HPOS].z, c[2], v[OPOS];
  DP4   o[HPOS].w, c[3], v[OPOS];
  DP3   R0.x, c[4], v[NRML];
  DP3   R0.y, c[5], v[NRML];
  DP3   R0.z, c[6], v[NRML];          # R0 = n' = transformed normal
  DP3   R1.x, c[32], R0;              # R1.x = Lpos DOT n'
  DP3   R1.y, c[33], R0;              # R1.y = hHat DOT n'
  MOV   R1.w, c[38].x;                # R1.w = specular power
  LIT   R2, R1;                       # Compute lighting values
  MAD   R3, c[35].x, R2.y, c[35].y;   # diffuse + emissive
  MAD   o[COL0].xyz, c[36], R2.z, R3; # + specular
  END
```

Can you perturb transformed vertex positions with a vertex program?

Yes. Here is an example that performs an object-space diffuse lighting computations and perturbs the vertex position based on this lighting result. Do not take this example too seriously.

```
!!VP1.0
#
# c[0-3]  = modelview projection (composite) matrix
# c[32]   = normalized light direction in object-space
# c[35]   = yellow diffuse material, (1.0, 1.0, 0.0, 1.0)
# c[64].x = 0.0
# c[64].z = 0.125, a scaling factor
#
# outputs diffuse illumination for color and perturbed position
#
DP3   R0, c[32], v[NRML];     # light direction DOT normal
MUL   o[COL0].xyz, R0, c[35];
MAX   R0, c[64].x, R0;
MUL   R0, R0, v[NRML];
MUL   R0, R0, c[64].z;
ADD   R1, v[OPOS], -R0;       # perturb object space position
DP4   o[HPOS].x, c[0], R1;
DP4   o[HPOS].y, c[1], R1;
DP4   o[HPOS].z, c[2], R1;
DP4   o[HPOS].w, c[3], R1;
END
```

What if more exponential precision is needed than provided by the builtin EXP instruction?

A sequence of vertex program instructions can be used refine the initial EXP approximation. The pseudo-macro below shows an example of how to refine the EXP approximation.

The psuedo-macro requires 10 instructions, 1 temp register, and 2 constant locations.

```
CE0 = { 9.61597636e-03, -1.32823968e-03, 1.47491097e-04, -1.08635004e-05 };
CE1 = { 1.00000000e+00, -6.93147182e-01, 2.40226462e-01, -5.55036440e-02 };

/* Rt != Ro && Rt != Ri */
EXP_MACRO(Ro:vector, Ri:scalar, Rt:vector) {
   EXP Rt, Ri.x;                  /* Use appropriate component of Ri */
   MAD Rt.w, c[CE0].w, Rt.y, c[CE0].z;
   MAD Rt.w, Rt.w,Rt.y, c[CE0].y;
   MAD Rt.w, Rt.w,Rt.y, c[CE0].x;
   MAD Rt.w, Rt.w,Rt.y, c[CE1].w;
   MAD Rt.w, Rt.w,Rt.y, c[CE1].z;
   MAD Rt.w, Rt.w,Rt.y, c[CE1].y;
   MAD Rt.w, Rt.w,Rt.y, c[CE1].x;
   RCP Rt.w, Rt.w;
   MUL Ro, Rt.w, Rt.x;            /* Apply user write mask to Ro */
}
```

Simulation gives |max abs error| < 3.77e-07 over the range (0.0 <= x < 1.0). Actual vertex program precision may be slightly less accurate than this.

What if more exponential precision is needed than provided by the builtin LOG instruction?

The pseudo-macro requires 10 instructions, 1 temp register, and 3 constant locations.

```
    CL0 = { 2.41873696e-01, -1.37531206e-01, 5.20646796e-02, -9.31049418e-03 };
    CL1 = { 1.44268966e+00, -7.21165776e-01, 4.78684813e-01, -3.47305417e-01 };
    CL2 = { 1.0, NA, NA, NA };

    /* Rt != Ro && Rt != Ri */
    LOG_MACRO(Ro:vector, Ri:scalar, Rt:vector) {
       LOG Rt, Ri.x;                  /* Use appropriate component of Ri */
       ADD Rt.y, Rt.y, -c[CL2].x;
       MAD Rt.w, c[CL0].w, Rt.y, c[CL0].z;
       MAD Rt.w, Rt.w, Rt.y,c[CL0].y;
       MAD Rt.w, Rt.w, Rt.y,c[CL0].x;
       MAD Rt.w, Rt.w, Rt.y,c[CL1].w;
       MAD Rt.w, Rt.w, Rt.y,c[CL1].z;
       MAD Rt.w, Rt.w, Rt.y,c[CL1].y;
       MAD Rt.w, Rt.w, Rt.y,c[CL1].x;
       MAD Ro, Rt.w, Rt.y, Rt.x;      /* Apply user write mask to Ro */
    }
```

Simulation gives |max abs error| < 1.79e-07 over the range (1.0 <= x < 2.0). Actual vertex program precision may be slightly less accurate than this.

New Procedures and Functions

```
    void BindProgramNV(enum target, uint id);

    void DeleteProgramsNV(sizei n, const uint *ids);

    void ExecuteProgramNV(enum target, uint id, const float *params);

    void GenProgramsNV(sizei n, uint *ids);

    boolean AreProgramsResidentNV(sizei n, const uint *ids,
                                  boolean *residences);

    void RequestResidentProgramsNV(sizei n, uint *ids);

    void GetProgramParameterfvNV(enum target, uint index,
                                 enum pname, float *params);
    void GetProgramParameterdvNV(enum target, uint index,
                                 enum pname, double *params);

    void GetProgramivNV(uint id, enum pname, int *params);

    void GetProgramStringNV(uint id, enum pname, ubyte *program);

    void GetTrackMatrixivNV(enum target, uint address,
                            enum pname, int *params);
    void GetVertexAttribdvNV(uint index, enum pname, double *params);
    void GetVertexAttribfvNV(uint index, enum pname, float *params);
    void GetVertexAttribivNV(uint index, enum pname, int *params);

    void GetVertexAttribPointervNV(uint index, enum pname, void **pointer);

    boolean IsProgramNV(uint id);

    void LoadProgramNV(enum target, uint id, sizei len,
                       const ubyte *program);

    void ProgramParameter4fNV(enum target, uint index,
                              float x, float y, float z, float w)
    void ProgramParameter4dNV(enum target, uint index,
                              double x, double y, double z, double w)

    void ProgramParameter4dvNV(enum target, uint index,
                               const double *params);
    void ProgramParameter4fvNV(enum target, uint index,
                               const float *params);

    void ProgramParameters4dvNV(enum target, uint index,
                                uint num, const double *params);
    void ProgramParameters4fvNV(enum target, uint index,
                                uint num, const float *params);

    void TrackMatrixNV(enum target, uint address,
                       enum matrix, enum transform);

    void VertexAttribPointerNV(uint index, int size, enum type, sizei stride,
                               const void *pointer);

    void VertexAttrib1sNV(uint index, short x);
    void VertexAttrib1fNV(uint index, float x);
    void VertexAttrib1dNV(uint index, double x);
    void VertexAttrib2sNV(uint index, short x, short y);
    void VertexAttrib2fNV(uint index, float x, float y);
    void VertexAttrib2dNV(uint index, double x, double y);
    void VertexAttrib3sNV(uint index, short x, short y, short z);
    void VertexAttrib3fNV(uint index, float x, float y, float z);
    void VertexAttrib3dNV(uint index, double x, double y, double z);
    void VertexAttrib4sNV(uint index, short x, short y, short z, short w);
    void VertexAttrib4fNV(uint index, float x, float y, float z, float w);
    void VertexAttrib4dNV(uint index, double x, double y, double z, double w);
    void VertexAttrib4ubNV(uint index, ubyte x, ubyte y, ubyte z, ubyte w);
    void VertexAttrib1svNV(uint index, const short *v);
    void VertexAttrib1fvNV(uint index, const float *v);
    void VertexAttrib1dvNV(uint index, const double *v);
    void VertexAttrib2svNV(uint index, const short *v);
    void VertexAttrib2fvNV(uint index, const float *v);
    void VertexAttrib2dvNV(uint index, const double *v);
    void VertexAttrib3svNV(uint index, const short *v);
    void VertexAttrib3fvNV(uint index, const float *v);
    void VertexAttrib3dvNV(uint index, const double *v);
    void VertexAttrib4svNV(uint index, const short *v);
    void VertexAttrib4fvNV(uint index, const float *v);
    void VertexAttrib4dvNV(uint index, const double *v);
    void VertexAttrib4ubvNV(uint index, const ubyte *v);

    void VertexAttribs1svNV(uint index, sizei n, const short *v);
    void VertexAttribs1fvNV(uint index, sizei n, const float *v);
    void VertexAttribs1dvNV(uint index, sizei n, const double *v);
    void VertexAttribs2svNV(uint index, sizei n, const short *v);
    void VertexAttribs2fvNV(uint index, sizei n, const float *v);
    void VertexAttribs2dvNV(uint index, sizei n, const double *v);
    void VertexAttribs3svNV(uint index, sizei n, const short *v);
    void VertexAttribs3fvNV(uint index, sizei n, const float *v);
    void VertexAttribs3dvNV(uint index, sizei n, const double *v);
    void VertexAttribs4svNV(uint index, sizei n, const short *v);
    void VertexAttribs4fvNV(uint index, sizei n, const float *v);
    void VertexAttribs4dvNV(uint index, sizei n, const double *v);
    void VertexAttribs4ubvNV(uint index, sizei n, const ubyte *v);
```

New Tokens

```
    Accepted by the <cap> parameter of Disable, Enable, and IsEnabled,
    and by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv,
    and GetDoublev, and by the <target> parameter of BindProgramNV,
    ExecuteProgramNV, GetProgramParameter[df]vNV, GetTrackMatrixivNV,
    LoadProgramNV, ProgramParameter[s]4[df][v]NV, and TrackMatrixNV:

        VERTEX_PROGRAM_NV                              0x8620

    Accepted by the <cap> parameter of Disable, Enable, and IsEnabled,
    and by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv,
    and GetDoublev:

        VERTEX_PROGRAM_POINT_SIZE_NV                   0x8642
        VERTEX_PROGRAM_TWO_SIDE_NV                     0x8643

    Accepted by the <target> parameter of ExecuteProgramNV and
    LoadProgramNV:

        VERTEX_STATE_PROGRAM_NV                        0x8621

    Accepted by the <pname> parameter of GetVertexAttrib[dfi]vNV:

        ATTRIB_ARRAY_SIZE_NV                           0x8623
        ATTRIB_ARRAY_STRIDE_NV                         0x8624
        ATTRIB_ARRAY_TYPE_NV                           0x8625
        CURRENT_ATTRIB_NV                              0x8626
    Accepted by the <pname> parameter of GetProgramParameterfvNV
    and GetProgramParameterdvNV:

        PROGRAM_PARAMETER_NV                            0x8644

    Accepted by the <pname> parameter of GetVertexAttribPointervNV:

        ATTRIB_ARRAY_POINTER_NV                         0x8645

    Accepted by the <pname> parameter of GetProgramivNV:

        PROGRAM_TARGET_NV                               0x8646
        PROGRAM_LENGTH_NV                               0x8627
        PROGRAM_RESIDENT_NV                             0x8647

    Accepted by the <pname> parameter of GetProgramStringNV:

        PROGRAM_STRING_NV                               0x8628

    Accepted by the <pname> parameter of GetTrackMatrixivNV:

        TRACK_MATRIX_NV                                 0x8648
        TRACK_MATRIX_TRANSFORM_NV                       0x8649

    Accepted by the <pname> parameter of GetBooleanv, GetIntegerv,
    GetFloatv, and GetDoublev:

        MAX_TRACK_MATRIX_STACK_DEPTH_NV                 0x862E
        MAX_TRACK_MATRICES_NV                           0x862F
        CURRENT_MATRIX_STACK_DEPTH_NV                   0x8640
        CURRENT_MATRIX_NV                               0x8641
        VERTEX_PROGRAM_BINDING_NV                       0x864A
        PROGRAM_ERROR_POSITION_NV                       0x864B

    Accepted by the <matrix> parameter of TrackMatrixNV:

        NONE
        MODELVIEW
        PROJECTION
        TEXTURE
        COLOR (if ARB_imaging is supported)
        MODELVIEW_PROJECTION_NV                         0x8629
        TEXTUREi_ARB

    where i is between 0 and n-1 where n is the number of texture units
    supported.
    Accepted by the <matrix> parameter of TrackMatrixNV and by the
    <mode> parameter of MatrixMode:

        MATRIX0_NV                                      0x8630
        MATRIX1_NV                                      0x8631
        MATRIX2_NV                                      0x8632
        MATRIX3_NV                                      0x8633
        MATRIX4_NV                                      0x8634
        MATRIX5_NV                                      0x8635
        MATRIX6_NV                                      0x8636
        MATRIX7_NV                                      0x8637

        (Enumerants 0x8638 through 0x863F are reserved for further matrix
        enumerants 8 through 15.)

    Accepted by the <transform> parameter of TrackMatrixNV:

        IDENTITY_NV                                     0x862A
        INVERSE_NV                                      0x862B
        TRANSPOSE_NV                                    0x862C
        INVERSE_TRANSPOSE_NV                            0x862D

    Accepted by the <array> parameter of EnableClientState and
    DisableClientState, by the <cap> parameter of IsEnabled, and by
    the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and
    GetDoublev:

        VERTEX_ATTRIB_ARRAY0_NV                         0x8650
        VERTEX_ATTRIB_ARRAY1_NV                         0x8651
        VERTEX_ATTRIB_ARRAY2_NV                         0x8652
        VERTEX_ATTRIB_ARRAY3_NV                         0x8653
        VERTEX_ATTRIB_ARRAY4_NV                         0x8654
        VERTEX_ATTRIB_ARRAY5_NV                         0x8655
        VERTEX_ATTRIB_ARRAY6_NV                         0x8656
        VERTEX_ATTRIB_ARRAY7_NV                         0x8657
        VERTEX_ATTRIB_ARRAY8_NV                         0x8658
        VERTEX_ATTRIB_ARRAY9_NV                         0x8659
        VERTEX_ATTRIB_ARRAY10_NV                        0x865A
        VERTEX_ATTRIB_ARRAY11_NV                        0x865B
        VERTEX_ATTRIB_ARRAY12_NV                        0x865C
        VERTEX_ATTRIB_ARRAY13_NV                        0x865D
        VERTEX_ATTRIB_ARRAY14_NV                        0x865E
        VERTEX_ATTRIB_ARRAY15_NV                        0x865F
```

    Accepted by the <target> parameter of GetMapdv, GetMapfv, GetMapiv,
    Map1d and Map1f and by the <cap> parameter of Enable, Disable, and
    IsEnabled, and by the <pname> parameter of GetBooleanv, GetIntegerv,
    GetFloatv, and GetDoublev:

MAP1_VERTEX_ATTRIB0_4_NV	0x8660
MAP1_VERTEX_ATTRIB1_4_NV	0x8661
MAP1_VERTEX_ATTRIB2_4_NV	0x8662
MAP1_VERTEX_ATTRIB3_4_NV	0x8663
MAP1_VERTEX_ATTRIB4_4_NV	0x8664
MAP1_VERTEX_ATTRIB5_4_NV	0x8665
MAP1_VERTEX_ATTRIB6_4_NV	0x8666
MAP1_VERTEX_ATTRIB7_4_NV	0x8667
MAP1_VERTEX_ATTRIB8_4_NV	0x8668
MAP1_VERTEX_ATTRIB9_4_NV	0x8669
MAP1_VERTEX_ATTRIB10_4_NV	0x866A
MAP1_VERTEX_ATTRIB11_4_NV	0x866B
MAP1_VERTEX_ATTRIB12_4_NV	0x866C
MAP1_VERTEX_ATTRIB13_4_NV	0x866D
MAP1_VERTEX_ATTRIB14_4_NV	0x866E
MAP1_VERTEX_ATTRIB15_4_NV	0x866F

    Accepted by the <target> parameter of GetMapdv, GetMapfv, GetMapiv,
    Map2d and Map2f and by the <cap> parameter of Enable, Disable, and
    IsEnabled, and by the <pname> parameter of GetBooleanv, GetIntegerv,
    GetFloatv, and GetDoublev:

MAP2_VERTEX_ATTRIB0_4_NV	0x8670
MAP2_VERTEX_ATTRIB1_4_NV	0x8671
MAP2_VERTEX_ATTRIB2_4_NV	0x8672
MAP2_VERTEX_ATTRIB3_4_NV	0x8673
MAP2_VERTEX_ATTRIB4_4_NV	0x8674
MAP2_VERTEX_ATTRIB5_4_NV	0x8675
MAP2_VERTEX_ATTRIB6_4_NV	0x8676
MAP2_VERTEX_ATTRIB7_4_NV	0x8677
MAP2_VERTEX_ATTRIB8_4_NV	0x8678
MAP2_VERTEX_ATTRIB9_4_NV	0x8679
MAP2_VERTEX_ATTRIB10_4_NV	0x867A
MAP2_VERTEX_ATTRIB11_4_NV	0x867B
MAP2_VERTEX_ATTRIB12_4_NV	0x867C
MAP2_VERTEX_ATTRIB13_4_NV	0x867D
MAP2_VERTEX_ATTRIB14_4_NV	0x867E
MAP2_VERTEX_ATTRIB15_4_NV	0x867F

Additions to Chapter 2 of the OpenGL 1.2.1 Specification (OpenGL Operation)

-- Section 2.10 "Coordinate Transformations"

Add this initial discussion:

"Per-vertex parameters are transformed before the transformation
results are used to generate primitives for rasterization, establish
a raster position, or generate vertices for selection or feedback.

Each vertex's per-vertex parameters are transformed by one of
two vertex transformation modes. The first vertex transformation mode
is GL's conventional vertex transformation model. The second mode,

known as 'vertex program' mode, transforms the vertex's per-vertex parameters by an application-supplied vertex program.

Vertex program mode is enabled and disabled, respectively, by

```
  void Enable(enum target);
```

and

```
  void Disable(enum target);
```

with target equal to VERTEX_PROGRAM_NV. When vertex program mode is enabled, vertices are transformed by the currently bound vertex program as discussed in section 2.14."

Update the original initial paragraph in the section to read:

"When vertex program mode is disabled, vertices, normals, and texture coordinates are transformed before their coordinates are used to produce an image in the framebuffer. We begin with a description of how vertex coordinates are transformed and how the transformation is controlled in the case when vertex program mode is disabled. The discussion that continues through section 2.13 applies when vertex program mode is disabled."

-- Section 2.10.2 "Matrices"

Change the first paragraph to read:

"The projection matrix and model-view matrix are set and modified with a variety of commands. The affected matrix is determined by the current matrix mode. The current matrix mode is set with

```
  void MatrixMode(enum mode);
```

which takes one of the pre-defined constants TEXTURE, MODELVIEW, COLOR, PROJECTION, or MATRIXi_NV as the argument. In the case of MATRIXi_NV, i is an integer between 0 and n-1 indicating one of n tracking matrices where n is the value of the implementation defined constant MAX_TRACK_MATRICES_NV. TEXTURE is described later in section 2.10.2, and COLOR is described in section 3.6.3. The tracking matrices of the form MATRIXi_NV are described in section 2.14.5. If the current matrix mode is MODELVIEW, then matrix operations apply to the model-view matrix; if PROJECTION, then they apply to the projection matrix."

Change the last paragraph to read:

"The state required to implement transformations consists of a n-value integer indicating the current matrix mode (where n is 4 + the number of tracking matrices supported), a stack of at least two 4x4 matrices for each of COLOR, PROJECTION, and TEXTURE with associated stack pointers, n stacks (where n is at least 8) of at least one 4x4 matrix for each MATRIXi_NV with associated stack pointers, and a stack of at least 32 4x4 matrices with an associated stack pointer for MODELVIEW. Initially, there is only one matrix on each stack, and all matrices are set to the identity. The initial matrix mode is MODELVIEW."

-- NEW Section 2.14 "Vertex Programs"

"The conventional GL vertex transformation model described in sections 2.10 through 2.13 is a configurable but essentially hard-wired sequence of per-vertex computations based on a canonical set of per-vertex parameters and vertex transformation related state such as transformation matrices, lighting parameters, and texture coordinate generation parameters.

The general success and utility of the conventional GL vertex transformation model reflects its basic correspondence to the typical vertex transformation requirements of 3D applications.

However when the conventional GL vertex transformation model is not sufficient, the vertex program mode provides a substantially more flexible model for vertex transformation. The vertex program mode permits applications to define their own vertex programs.

2.14.1 The Vertex Program Execution Model

A vertex program is a sequence of floating-point 4-component vector operations that operate on per-vertex attributes and program parameters. Vertex programs execute on a per-vertex basis and operate on each vertex completely independently from the processing of other vertices. Vertex programs execute a finite fixed sequence of instructions with no branching or looping. Vertex programs execute without data hazards so results computed in one operation can be used immediately afterwards. The result of a vertex program is a set of vertex result vectors that becomes the transformed vertex parameters used by primitive assembly.

Vertex programs use a specific well-defined instruction set, register set, and operational model defined in the following sections.

The vertex program register set consists of five types of registers described in the following five sections.

2.14.1.1 The Vertex Attribute Registers

The Vertex Attribute Registers are sixteen 4-component vector floating-point registers containing the current vertex's per-vertex attributes. These registers are numbered 0 through 15. These registers are private to each vertex program invocation and are initialized at each vertex program invocation by the current vertex attribute state specified with VertexAttribNV commands. These registers are read-only during vertex program execution. The VertexAttribNV commands used to update the vertex attribute registers can be issued both outside and inside of Begin/End pairs. Vertex program execution is provoked by updating vertex attribute zero. Updating vertex attribute zero outside of a Begin/End pair is ignored without generating any error (identical to the Vertex command operation).

The commands

```
  void VertexAttrib{1234}{sfd}NV(uint index, T coords);
  void VertexAttrib{1234}{sfd}vNV(uint index, T coords);
  void VertexAttrib4ubNV(uint index, T coords);
  void VertexAttrib4ubvNV(uint index, T coords);
```

specify the particular current vertex attribute indicated by index. The coordinates for each vertex attribute are named x, y, z, and w. The VertexAttrib1NV family of commands sets the x coordinate to the provided single argument while setting y and z to 0 and w to 1. Similarly, VertexAttrib2NV sets x and y to the specified values, z to 0 and w to 1; VertexAttrib3NV sets x, y, and z, with w set to 1, and VertexAttrib4NV sets all four coordinates. The error INVALID_VALUE is generated if index is greater than 15.

No conversions are applied to the vertex attributes specified as type short, float, or double. However, vertex attributes specified as type ubyte are converted as described by Table 2.6.

The commands

```
  void VertexAttribs{1234}{sfd}vNV(uint index, sizei n, T coords[]);
  void VertexAttribs4ubvNV(uint index, sizei n, GLubyte coords[]);
```

specify a contiguous set of n vertex attributes. The effect of

```
  VertexAttribs{1234}{sfd}vNV(index, n, coords)
```

is the same (assuming no errors) as the command sequence

```
  #define NUM k  /* where k is 1, 2, 3, or 4 components */
  int i;
  for (i=n-1; i>=0; i--) {
    VertexAttrib{NUM}{sfd}vNV(i+index, &coords[i*NUM]);
  }
```

VertexAttribs4ubvNV behaves similarly.

The VertexAttribNV calls equivalent to VertexAttribsNV are issued in reverse order so that vertex program execution is provoked when index is zero only after all the other vertex attributes have first been specified.

2.14.1.2 The Program Parameter Registers

The Program Parameter Registers are ninety-six 4-component floating-point vector registers containing the vertex program parameters. These registers are numbered 0 through 95. This relatively large set of registers is intended to hold parameters such as matrices, lighting parameters, and constants required by vertex programs. Vertex program parameter registers can be updated in one of two ways: by the ProgramParameterNV commands outside of a Begin/End pair or by a vertex state program executed outside of a Begin/End pair (vertex state programs are discussed in section 2.14.3).

The commands

```
  void ProgramParameter4fNV(enum target, uint index,
                            float x, float y, float z, float w)
  void ProgramParameter4dNV(enum target, uint index,
                            double x, double y, double z, double w)
```

specify the particular program parameter indicated by index.
The coordinates values x, y, z, and w are assigned to the respective
components of the particular program parameter. target must be
VERTEX_PROGRAM_NV.

The commands

```
  void ProgramParameter4dvNV(enum target, uint index, double *params);
  void ProgramParameter4fvNV(enum target, uint index, float *params);
```

operate identically to ProgramParameter4fNV and ProgramParameter4dNV
respectively except that the program parameters are passed as an
array of four components.

The commands

```
  void ProgramParameters4dvNV(enum target, uint index,
                              uint num, double *params);
  void ProgramParameters4fvNV(enum target, uint index,
                              uint num, float *params);
```

specify a contiguous set of num program parameters. target must
be VERTEX_PROGRAM_NV. The effect is the same (assuming no errors) as

```
  for (i=index; i<index+num; i++) {
    ProgramParameter4{fd}vNV(target, i, &params[i*4]);
  }
```

The program parameter registers are shared to all vertex program
invocations within a rendering context. ProgramParameterNV command
updates and vertex state program executions are serialized with
respect to vertex program invocations and other vertex state program
executions.

Writes to the program parameter registers during vertex state program
execution can be maskable on a per-component basis.

The error INVALID_VALUE is generated if any ProgramParameterNV has
an index is greater than 95.

The initial value of all ninety-six program parameter registers is
(0,0,0,0).

2.14.1.3 The Address Register

The Address Register is a single 4-component vector signed 32-bit
integer register though only the x component of the vector is
accessible. The register is private to each vertex program invocation
and is initialized to (0,0,0,0) at every vertex program invocation.
This register can be written during vertex program execution (but

not read) and its value can be used for as a relative offset for reading vertex program parameter registers. Only the vertex program parameter registers can be read using relative addressing (writes using relative addressing are not supported).

See the discussion of relative addressing of program parameters in section 2.14.1.9 and the discussion of the ARL instruction in section 2.14.1.10.1.

2.14.1.4 The Temporary Registers

The Temporary Registers are twelve 4-component floating-point vector registers used to hold temporary results during vertex program execution. These registers are numbered 0 through 11. These registers are private to each vertex program invocation and initialized to (0,0,0,0) at every vertex program invocation. These registers can be read and written during vertex program execution. Writes to these registers can be maskable on a per-component basis.

2.14.1.5 The Vertex Result Register Set

The Vertex Result Registers are fifteen 4-component floating-point vector registers used to write the results of a vertex program. Each register value is initialized to (0,0,0,1) at the invocation of each vertex program. Writes to the vertex result registers can be maskable on a per-component basis. These registers are named in Table X.1 and further discussed below.

Vertex Result Register Name	Description	Component Interpretation
HPOS	Homogeneous clip space position	(x,y,z,w)
COL0	Primary color (front-facing)	(r,g,b,a)
COL1	Secondary color (front-facing)	(r,g,b,a)
BFC0	Back-facing primary color	(r,g,b,a)
BFC1	Back-facing secondary color	(r,g,b,a)
FOGC	Fog coordinate	(f,*,*,*)
PSIZ	Point size	(p,*,*,*)
TEX0	Texture coordinate set 0	(s,t,r,q)
TEX1	Texture coordinate set 1	(s,t,r,q)
TEX2	Texture coordinate set 2	(s,t,r,q)
TEX3	Texture coordinate set 3	(s,t,r,q)
TEX4	Texture coordinate set 4	(s,t,r,q)
TEX5	Texture coordinate set 5	(s,t,r,q)
TEX6	Texture coordinate set 6	(s,t,r,q)
TEX7	Texture coordinate set 7	(s,t,r,q)

Table X.1: Vertex Result Registers.

HPOS is the transformed vertex's homogeneous clip space position. The vertex's homogeneous clip space position is converted to normalized device coordinates and transformed to window coordinates as described at the end of section 2.10 and in section 2.11. Further processing (subsequent to vertex program termination) is responsible for clipping primitives assembled from vertex program-generated vertices as described in section 2.10 but all client-defined clip planes are treated as if they are disabled when vertex program mode is enabled.

Four distinct color results can be generated for each vertex. COL0 is the transformed vertex's front-facing primary color. COL1 is the transformed vertex's front-facing secondary color. BFC0 is the transformed vertex's back-facing primary color. BFC1 is the transformed vertex's back-facing secondary color.

Primitive coloring may operate in two-sided color mode. This behavior is enabled and disabled by calling Enable or Disable with the symbolic value VERTEX_PROGRAM_TWO_SIDE_NV. The selection between the back-facing colors and the front-facing colors depends on the primitive of which the vertex is a part. If the primitive is a point or a line segment, the front-facing colors are always selected. If the primitive is a polygon and two-sided color mode is disabled, the front-facing colors are selected. If it is a polygon and two-sided color mode is enabled, then the selection is based on the sign of the (clipped or unclipped) polygon's signed area computed in window coordinates. This facingness determination is identical to the two-sided lighting facingness determination described in section 2.13.1.

The selected primary and secondary colors for each primitive are clamped to the range [0,1] and then interpolated across the assembled primitive during rasterization with at least 8-bit accuracy for each color component.

FOGC is the transformed vertex's fog coordinate. The register's first floating-point component is interpolated across the assembled primitive during rasterization and used as the fog distance to compute per-fragment the fog factor when fog is enabled. However, if both fog and vertex program mode are enabled, but the FOGC vertex result register is not written, the fog factor is overridden to 1.0. The register's other three components are ignored.

Point size determination may operate in program-specified point size mode. This behavior is enabled and disabled by calling Enable or Disable with the symbolic value VERTEX_PROGRAM_POINT_SIZE_NV. If the vertex is for a point primitive and the mode is enabled and the PSIZ vertex result is written, the point primitive's size is determined by the clamped x component of the PSIZ register. Otherwise (because vertex program mode is disabled, program-specified point size mode is disabled, or because the vertex program did not write PSIZ), the point primitive's size is determined by the point size state (the state specified using the PointSize command).

The PSIZ register's x component is clamped to the range zero through either the hi value of ALIASED_POINT_SIZE_RANGE if point smoothing is disabled or the hi value of the SMOOTH_POINT_SIZE_RANGE if point smoothing is enabled. The register's other three components are ignored.

If the vertex is not for a point primitive, the value of the PSIZ vertex result register is ignored.

TEX0 through TEX7 are the transformed vertex's texture coordinate sets for texture units 0 through 7. These floating-point coordinates are interpolated across the assembled primitive during rasterization and used for accessing textures. If the number of texture units supported is less than eight, the values of vertex result registers that do not correspond to existent texture units are ignored.

2.14.1.6 Semantic Meaning for Vertex Attributes and Program Parameters

One important distinction between the conventional GL vertex transformation mode and the vertex program mode is that per-vertex parameters and other state parameters in vertex program mode do not have dedicated semantic interpretations the way that they do with the conventional GL vertex transformation mode.

For example, in the conventional GL vertex transformation mode, the Normal command specifies a per-vertex normal. The semantic that the Normal command supplies a normal for lighting is established because that is how the per-vertex attribute supplied by the Normal command is used by the conventional GL vertex transformation mode. Similarly, other state parameters such as a light source position have semantic interpretations based on how the conventional GL vertex transformation model uses each particular parameter.

In contrast, vertex attributes and program parameters for vertex programs have no pre-defined semantic meanings. The meaning of a vertex attribute or program parameter in vertex program mode is defined by how the vertex attribute or program parameter is used by the current vertex program to compute and write values to the Vertex Result Registers. This is the reason that per-vertex attributes and program parameters for vertex programs are numbered instead of named.

For convenience however, the existing per-vertex parameters for the conventional GL vertex transformation mode (vertices, normals, colors, fog coordinates, vertex weights, and texture coordinates) are aliased to numbered vertex attributes. This aliasing is specified in Table X.2. The table includes how the various conventional components map to the 4-component vertex attribute components.

Vertex Attribute Register Number	Conventional Per-vertex Parameter	Conventional Per-vertex Parameter Command	Conventional Component Mapping
0	vertex position	Vertex	x,y,z,w
1	vertex weights	VertexWeightEXT	w,0,0,1
2	normal	Normal	x,y,z,1
3	primary color	Color	r,g,b,a
4	secondary color	SecondaryColorEXT	r,g,b,1
5	fog coordinate	FogCoordEXT	fc,0,0,1
6	-	-	-
7	-	-	-
8	texture coord 0	MultiTexCoord(GL_TEXTURE0_ARB, ...)	s,t,r,q
9	texture coord 1	MultiTexCoord(GL_TEXTURE1_ARB, ...)	s,t,r,q
10	texture coord 2	MultiTexCoord(GL_TEXTURE2_ARB, ...)	s,t,r,q
11	texture coord 3	MultiTexCoord(GL_TEXTURE3_ARB, ...)	s,t,r,q
12	texture coord 4	MultiTexCoord(GL_TEXTURE4_ARB, ...)	s,t,r,q
13	texture coord 5	MultiTexCoord(GL_TEXTURE5_ARB, ...)	s,t,r,q
14	texture coord 6	MultiTexCoord(GL_TEXTURE6_ARB, ...)	s,t,r,q
15	texture coord 7	MultiTexCoord(GL_TEXTURE7_ARB, ...)	s,t,r,q

Table X.2: Aliasing of vertex attributes with conventional per-vertex parameters.

Only vertex attribute zero is treated specially because it is the attribute that provokes the execution of the vertex program; this is the attribute that aliases to the Vertex command's vertex coordinates.

The result of a vertex program is the set of post-transformation vertex parameters written to the Vertex Result Registers. All vertex programs must write a homogeneous clip space position, but the other Vertex Result Registers can be optionally written.

Clipping and culling are not the responsibility of vertex programs because these operations assume the assembly of multiple vertices into a primitive. View frustum clipping is performed subsequent to vertex program execution. Clip planes are not supported in vertex program mode.

2.14.1.7 Vertex Program Specification

Vertex programs are specified as an array of ubytes. The array is a string of ASCII characters encoding the program.

The command

```
  LoadProgramNV(enum target, uint id, sizei len,
                const ubyte *program);
```

loads a vertex program when the target parameter is VERTEX_PROGRAM_NV. Multiple programs can be loaded with different names. id names the program to load. The name space for programs is the positive integers (zero is reserved). The error INVALID_VALUE occurs if a program is loaded with an id of zero. The error INVALID_OPERATION is generated if a program is loaded for an id that is currently loaded with a program of a different program target. Managing the program name space and binding to vertex programs is discussed later in section 2.14.1.8.

program is a pointer to an array of ubytes that represents the program being loaded. The length of the array is indicated by len.

A second program target type known as vertex state programs is discussed in 2.14.4.

At program load time, the program is parsed into a set of tokens possibly separated by white space. Spaces, tabs, newlines, carriage returns, and comments are considered whitespace. Comments begin with the character "#" and are terminated by a newline, a carriage return, or the end of the program array.

The Backus-Naur Form (BNF) grammar below specifies the syntactically valid sequences for vertex programs. The set of valid tokens can be inferred from the grammar. The token "" represents an empty string and is used to indicate optional rules. A program is invalid if it contains any undefined tokens or characters.

```
<program>              ::= "!!VP1.0" <instructionSequence> "END"

<instructionSequence>  ::= <instructionSequence> <instructionLine>
                         | <instructionLine>

<instructionLine>      ::= <instruction> ";"

<instruction>          ::= <ARL-instruction>
                         | <VECTORop-instruction>
                         | <SCALARop-instruction>
                         | <BINop-instruction>
                         | <TRIop-instruction>

<ARL-instruction>      ::= "ARL" <addrReg> "," <scalarSrcReg>

<VECTORop-instruction> ::= <VECTORop> <maskedDstReg> "," <swizzleSrcReg>

<SCALARop-instruction> ::= <SCALARop> <maskedDstReg> "," <scalarSrcReg>

<BINop-instruction>    ::= <BINop> <maskedDstReg> ","
                           <swizzleSrcReg> "," <swizzleSrcReg>

<TRIop-instruction>    ::= <TRIop> <maskedDstReg> ","
                           <swizzleSrcReg> "," <swizzleSrcReg> ","
                           <swizzleSrcReg>

<VECTORop>             ::= "MOV"
                         | "LIT"

<SCALARop>             ::= "RCP"
                         | "RSQ"
                         | "EXP"
                         | "LOG"
```

```
    <BINop>                 ::= "MUL"
                              | "ADD"
                              | "DP3"
                              | "DP4"
                              | "DST"
                              | "MIN"
                              | "MAX"
                              | "SLT"
                              | "SGE"

    <TRIop>                 ::= "MAD"

    <scalarSrcReg>          ::= <optionalSign> <srcReg> <scalarSuffix>

    <swizzleSrcReg>         ::= <optionalSign> <srcReg> <swizzleSuffix>

    <maskedDstReg>          ::= <dstReg> <optionalMask>

    <optionalMask>          ::= ""
                              | "." "x"
                              | "."     "y"
                              | "." "x" "y"
                              | "."         "z"
                              | "." "x"     "z"
                              | "."     "y" "z"
                              | "." "x" "y" "z"
                              | "."             "w"
                              | "." "x"         "w"
                              | "."     "y"     "w"
                              | "." "x" "y"     "w"
                              | "."         "z" "w"
                              | "." "x"     "z" "w"
                              | "."     "y" "z" "w"
                              | "." "x" "y" "z" "w"

    <optionalSign>          ::= "-"
                              | ""

    <srcReg>                ::= <vertexAttribReg>
                              | <progParamReg>
                              | <temporaryReg>

    <dstReg>                ::= <temporaryReg>
                              | <vertexResultReg>

    <vertexAttribReg>       ::= "v" "[" vertexAttribRegNum "]"
    <vertexAttribRegNum>   ::= decimal integer from 0 to 15 inclusive
                              | "OPOS"
                              | "WGHT"
                              | "NRML"
                              | "COL0"
                              | "COL1"
                              | "FOGC"
                              | "TEX0"
                              | "TEX1"
                              | "TEX2"
                              | "TEX3"
                              | "TEX4"
                              | "TEX5"
                              | "TEX6"
                              | "TEX7"

    <progParamReg>         ::= <absProgParamReg>
                              | <relProgParamReg>

    <absProgParamReg>      ::= "c" "[" <progParamRegNum> "]"

    <progParamRegNum>      ::= decimal integer from 0 to 95 inclusive

    <relProgParamReg>      ::= "c" "[" <addrReg> "]"
                              | "c" "[" <addrReg> "+" <progParamPosOffset> "]"
                              | "c" "[" <addrReg> "-" <progParamNegOffset> "]"

    <progParamPosOffset>   ::= decimal integer from 0 to 63 inclusive

    <progParamNegOffset>   ::= decimal integer from 0 to 64 inclusive

    <addrReg>              ::= "A0" "." "x"

    <temporaryReg>         ::= "R0"
                              | "R1"
                              | "R2"
                              | "R3"
                              | "R4"
                              | "R5"
                              | "R6"
                              | "R7"
                              | "R8"
                              | "R9"
                              | "R10"
                              | "R11"

    <vertexResultReg>      ::= "o" "[" vertexResultRegName "]"
    <vertexResultRegName>  ::= "HPOS"
                              | "COL0"
                              | "COL1"
                              | "BFC0"
                              | "BFC1"
                              | "FOGC"
                              | "PSIZ"
                              | "TEX0"
                              | "TEX1"
                              | "TEX2"
                              | "TEX3"
                              | "TEX4"
                              | "TEX5"
                              | "TEX6"
                              | "TEX7"

    <scalarSuffix>         ::= "." <component>

    <swizzleSuffix>        ::= ""
                              | "." <component>
                              | "." <component> <component>
                                    <component> <component>

    <component>            ::= "x"
                              | "y"
                              | "z"
                              | "w"
```

The <vertexAttribRegNum> rule matches both register numbers 0 through 15 and a set of mnemonics that abbreviate the aliasing of conventional the per-vertex parameters to vertex attribute register numbers. Table X.3 shows the mapping from mnemonic to vertex attribute register number and what the mnemonic abbreviates.

	Vertex Attribute	
Mnemonic	Register Number	Meaning
"OPOS"	0	object position
"WGHT"	1	vertex weight
"NRML"	2	normal
"COL0"	3	primary color
"COL1"	4	secondary color
"FOGC"	5	fog coordinate
"TEX0"	8	texture coordinate 0
"TEX1"	9	texture coordinate 1
"TEX2"	10	texture coordinate 2
"TEX3"	11	texture coordinate 3
"TEX4"	12	texture coordinate 4
"TEX5"	13	texture coordinate 5
"TEX6"	14	texture coordinate 6
"TEX7"	15	texture coordinate 7

Table X.3: The mapping between vertex attribute register numbers, mnemonics, and meanings.

A vertex programs fails to load if it does not write at least one component of the HPOS register.

A vertex program fails to load if it contains more than 128 instructions.

A vertex program fails to load if any instruction sources more than one unique program parameter register.

A vertex program fails to load if any instruction sources more than one unique vertex attribute register.

The error INVALID_OPERATION is generated if a vertex program fails to load because it is not syntactically correct or for one of the semantic restrictions listed above.

The error INVALID_OPERATION is generated if a program is loaded for id when id is currently loaded with a program of a different target.

A successfully loaded vertex program is parsed into a sequence of instructions. Each instruction is identified by its tokenized name. The operation of these instructions when executed is defined in section 2.14.1.10.

A successfully loaded program replaces the program previously assigned to the name specified by id. If the OUT_OF_MEMORY error is generated by LoadProgramNV, no change is made to the previous contents of the named program.

Querying the value of PROGRAM_ERROR_POSITION_NV returns a ubyte offset into the last loaded program string indicating where the first error in the program. If the program fails to load because of a semantic restriction that cannot be determined until the program is fully scanned, the error position will be len, the length of the program. If the program loads successfully, the value of PROGRAM_ERROR_POSITION_NV is assigned the value negative one.

2.14.1.8 Vertex Program Binding and Program Management

The current vertex program is invoked whenever vertex attribute zero is updated (whether by a VertexAttributeNV or Vertex command). The current vertex program is updated by

```
  BindProgramNV(enum target, uint id);
```

where target must be VERTEX_PROGRAM_NV. This binds the vertex program named by id as the current vertex program. The error INVALID_OPERATION is generated if id names a program that is not a vertex program (for example, if id names a vertex state program as described in section 2.14.4).

Binding to a nonexistent program id does not generate an error. In particular, binding to program id zero does not generate an error. However, because program zero cannot be loaded, program zero is always nonexistent. If a program id is successfully loaded with a new vertex program and id is also the currently bound vertex program, the new program is considered the currently bound vertex program.

The INVALID_OPERATION error is generated when both vertex program mode is enabled and Begin is called (or when a command that performs an implicit Begin is called) if the current vertex program is nonexistent or not valid. A vertex program may not be valid for reasons explained in section 2.14.5.

Programs are deleted by calling

```
  void DeleteProgramsNV(sizei n, const uint *ids);
```

ids contains n names of programs to be deleted. After a program is deleted, it becomes nonexistent, and its name is again unused. If a program that is currently bound is deleted, it is as though BindProgramNV has been executed with the same target as the deleted program and program zero. Unused names in ids are silently ignored, as is the value zero.

The command

```
  void GenProgramsNV(sizei n, uint *ids);
```

returns n previously unused program names in ids. These names are marked as used, for the purposes of GenProgramsNV only, but they become existent programs only when the are first loaded using LoadProgramNV. The error INVALID_VALUE is generated if n is negative.

An implementation may choose to establish a working set of programs on which binding and ExecuteProgramNV operations (execute programs are explained in section 2.14.4) are performed with higher performance. A program that is currently part of this working set is said to be resident.

The command

```
  boolean AreProgramsResidentNV(sizei n, const uint *ids,
                                boolean *residences);
```

returns TRUE if all of the n programs named in ids are resident, or if the implementation does not distinguish a working set. If at least one of the programs named in ids is not resident, then FALSE is returned, and the residence of each program is returned in residences. Otherwise the contents of residences are not changed. If any of the names in ids are nonexistent or zero, FALSE is returned, the error INVALID_VALUE is generated, and the contents of residences are indeterminate. The residence status of a single named program can also be queried by calling GetProgramivNV with id set to the name of the program and pname set to PROGRAM_RESIDENT_NV.

AreProgramsResidentNV indicates only whether a program is currently resident, not whether it could not be made resident. An implementation may choose to make a program resident only on first use, for example. The client may guide the GL implementation in determining which programs should be resident by requesting a set of programs to make resident.

The command

```
  void RequestResidentProgramsNV(sizei n, const uint *ids);
```

requests that the n programs named in ids should be made resident. While all the programs are not guaranteed to become resident, the implementation should make a best effort to make as many of the programs resident as possible. As a result of making the requested programs resident, program names not among the requested programs may become non-resident. Higher priority for residency should be given to programs listed earlier in the ids array. RequestResidentProgramsNV silently ignores attempts to make resident nonexistent program names or zero. AreProgramsResidentNV can be called after RequestResidentProgramsNV to determine which programs actually became resident.

2.14.1.9 Vertex Program Register Accesses

There are 17 vertex program instructions. The instructions and their respective input and output parameters are summarized in Table X.4.

Opcode	Inputs (scalar or vector)	Output (vector or replicated scalar)	Operation
ARL	s	address register	address register load
MOV	v	v	move
MUL	v,v	v	multiply
ADD	v,v	v	add
MAD	v,v,v	v	multiply and add
RCP	s	ssss	reciprocal
RSQ	s	ssss	reciprocal square root
DP3	v,v	ssss	3-component dot product
DP4	v,v	ssss	4-component dot product
DST	v,v	v	distance vector
MIN	v,v	v	minimum
MAX	v,v	v	maximum
SLT	v,v	v	set on less than
SGE	v,v	v	set on greater equal than
EXP	s	v	exponential base 2
LOG	s	v	logarithm base 2
LIT	v	v	light coefficients

Table X.4: Summary of vertex program instructions. "v" indicates a vector input or output, "s" indicates a scalar input, and "ssss" indicates a scalar output replicated across a 4-component vector.

Instructions use either scalar source values or swizzled source values, indicated in the grammar (see section 2.14.1.7) by the rules <scalarSrcReg> and <swizzleSrcReg> respectively. Either type of source value is negated when the <optionalSign> rule matches "-".

Scalar source register values select one of the source register's four components based on the <component> of the <scalarSuffix> rule. The characters "x", "y", "z", and "w" match the x, y, z, and w components respectively. The indicated component is used as a scalar for the particular source value.

Swizzled source register values may arbitrarily swizzle the source register's components based on the <swizzleSuffix> rule. In the case where the <swizzleSuffix> matches (ignoring whitespace) the pattern ".????" where each question mark is one of "x", "y", "z", or "w", this indicates the ith component of the source register value should come from the component named by the ith component in the sequence. For example, if the swizzle suffix is ".yzzx" and the source register contains [2.0, 8.0, 9.0, 0.0] the swizzled source register value used by the instruction is [8.0, 9.0, 9.0, 2.0].

If the <swizzleSuffix> rule matches "", this is treated the same as ".xyzw". If the <swizzleSuffix> rule matches (ignoring whitespace) ".x", ".y", ".z", or ".w", these are treated the same as ".xxxx", ".yyyy", ".zzzz", and ".wwww" respectively.

The register sourced for either a scalar source register value or a swizzled source register value is indicated in the grammar by the rule <srcReg>. The <vertexAttribReg>, <progParamReg>, and <temporaryReg> sub-rules correspond to one of the vertex attribute registers, program parameter registers, or temporary register respectively.

The vertex attribute and temporary registers are accessed absolutely based on the numbered register. In the case of vertex attribute registers, if the <vertexAttribRegNum> corresponds to a mnemonic, the corresponding register number from Table X.3 is used.

Either absolute or relative addressing can be used to access the program parameter registers. Absolute addressing is indicated by the grammar by the <absProgParamReg> rule. Absolute addressing accesses the numbered program parameter register indicated by the <progParamRegNum> rule. Relative addressing accesses the numbered program parameter register plus an offset. The offset is the positive value of <progParamPosOffset> if the <progParamPosOffset> rule is matched, or the offset is the negative value of <progParamNegOffset> if the <progParamNegOffset> rule is matched, or otherwise the offset is zero. Relative addressing is available only for program parameter registers and only for reads (not writes). Relative addressing reads outside of the 0 to 95 inclusive range always read the value (0,0,0,0).

The result of all instructions except ARL is written back to a masked destination register, indicated in the grammar by the rule <maskedDstReg>.

Writes to each component of the destination register can be masked, indicated in the grammar by the <optionalMask> rule. If the optional mask is "", all components are written. Otherwise, the optional mask names particular components to write. The characters "x", "y", "z", and "w" match the x, y, z, and w components respectively. For example, an optional mask of ".xzw" indicates that the x, z, and w components should be written but not the y component. The grammar requires that the destination register mask components must be listed in "xyzw" order.

The actual destination register is indicated in the grammar by the rule <dstReg>. The <temporaryReg> and <vertexResultReg>

sub-rules correspond to either the temporary registers or vertex result registers. The temporary registers are determined and accessed as described earlier.

The vertex result registers are accessed absolutely based on the named register. The <vertexResultRegName> rule corresponds to registers named in Table X.1.

2.14.1.10 Vertex Program Instruction Set Operations

The operation of the 17 vertex program instructions are described in this section. After the textual description of each instruction's operation, a register transfer level description is also presented.

The following conventions are used in each instruction's register transfer level description. The 4-component vector variables "t", "u", and "v" are assigned intermediate results. The destination register is called "destination". The three possible source registers are called "source0", "source1", and "source2" respectively.

The x, y, z, and w vector components are referred to with the suffixes ".x", ".y", ".z", and ".w" respectively. The suffix ".c" is used for scalar source register values and c represents the particular source register's selected scalar component. Swizzling of components is indicated with the suffixes ".c***", ".*c**", ".**c*", and ".***c" where c is meant to indicate the x, y, z, or w component selected for the particular source operand swizzle configuration. For example:

```
  t.x = source0.c***;
  t.y = source0.*c**;
  t.z = source0.**c*;
  t.w = source0.***c;
```

This example indicates that t should be assigned the swizzled version of the source0 operand based on the source0 operand's swizzle configuration.

The variables "negate0", "negate1", and "negate2" are booleans that are true when the respective source value should be negated. The variables "xmask", "ymask", "zmask", and "wmask" are booleans that are true when the destination write mask for the respective component is enabled for writing.

Otherwise, the register transfer level descriptions mimic ANSI C syntax.

The idiom "IEEE(expression)" represents the s23e8 single-precision result of the expression if evaluated using IEEE single-precision floating point operations. The IEEE idiom is used to specify the maximum allowed deviation from IEEE single-precision floating-point arithmetic results.

The following abbreviations are also used:

+Inf	floating-point representation of positive infinity
-Inf	floating-point representation of negative infinity
+NaN	floating-point representation of positive not a number
-NaN	floating-point representation of negative not a number
NA	not applicable or not used

2.14.1.10.1 ARL: Address Register Load

The ARL instruction moves value of the source scalar into the address register. Conceptually, the address register load instruction is a 4-component vector signed integer register, but the only valid address register component for writing and indexing is the x component. The only use for A0.x is as a base address for program parameter reads. The source value is a float that is truncated towards negative infinity into a signed integer.

```
    t.x = source0.c;
    if (negate0) t.x = -t.x;
    A0.x = floor(t.x);
```

2.14.1.10.2 MOV: Move

The MOV instruction moves the value of the source vector into the destination register.

```
    t.x = source0.c***;
    t.y = source0.*c**;
    t.z = source0.**c*;
    t.w = source0.***c;
    if (negate0) {
      t.x = -t.x;
      t.y = -t.y;
      t.z = -t.z;
      t.w = -t.w;
    }
    if (xmask) destination.x = t.x;
    if (ymask) destination.y = t.y;
    if (zmask) destination.z = t.z;
    if (wmask) destination.w = t.w;
```

2.14.1.10.3 MUL: Multiply

The MUL instruction multiplies the values of the two source vectors into the destination register.

```
    t.x = source0.c***;
    t.y = source0.*c**;
    t.z = source0.**c*;
    t.w = source0.***c;
    if (negate0) {
      t.x = -t.x;
      t.y = -t.y;
      t.z = -t.z;
      t.w = -t.w;
    }
    u.x = source1.c***;
    u.y = source1.*c**;
    u.z = source1.**c*;
    u.w = source1.***c;
    if (negate1) {
      u.x = -u.x;
      u.y = -u.y;
      u.z = -u.z;
      u.w = -u.w;
    }
    if (xmask) destination.x = t.x * u.x;
    if (ymask) destination.y = t.y * u.y;
    if (zmask) destination.z = t.z * u.z;
    if (wmask) destination.w = t.w * u.w;
```

2.14.1.10.4 ADD: Add

The ADD instruction adds the values of the two source vectors into the destination register.

```
    t.x = source0.c***;
    t.y = source0.*c**;
    t.z = source0.**c*;
    t.w = source0.***c;
    if (negate0) {
      t.x = -t.x;
      t.y = -t.y;
      t.z = -t.z;
      t.w = -t.w;
    }
    u.x = source1.c***;
    u.y = source1.*c**;
    u.z = source1.**c*;
    u.w = source1.***c;
    if (negate1) {
      u.x = -u.x;
      u.y = -u.y;
      u.z = -u.z;
      u.w = -u.w;
    }
    if (xmask) destination.x = t.x + u.x;
    if (ymask) destination.y = t.y + u.y;
    if (zmask) destination.z = t.z + u.z;
    if (wmask) destination.w = t.w + u.w;
```

2.14.1.10.5 MAD: Multiply and Add

The MAD instruction adds the value of the third source vector to the product of the values of the first and second two source vectors, writing the result to the destination register.

```
t.x = source0.c***;
t.y = source0.*c**;
t.z = source0.**c*;
t.w = source0.***c;
if (negate0) {
  t.x = -t.x;
  t.y = -t.y;
  t.z = -t.z;
  t.w = -t.w;
}
u.x = source1.c***;
u.y = source1.*c**;
u.z = source1.**c*;
u.w = source1.***c;
if (negate1) {
  u.x = -u.x;
  u.y = -u.y;
  u.z = -u.z;
  u.w = -u.w;
}
v.x = source2.c***;
v.y = source2.*c**;
v.z = source2.**c*;
v.w = source2.***c;
if (negate2) {
  v.x = -v.x;
  v.y = -v.y;
  v.z = -v.z;
  v.w = -v.w;
}
if (xmask) destination.x = t.x * u.x + v.x;
if (ymask) destination.y = t.y * u.y + v.y;
if (zmask) destination.z = t.z * u.z + v.z;
if (wmask) destination.w = t.w * u.w + v.w;
```

2.14.1.10.6 RCP: Reciprocal

The RCP instruction inverts the value of the source scalar into the destination register. The reciprocal of exactly 1.0 must be exactly 1.0.

Additionally the reciprocal of negative infinity gives [-0.0, -0.0, -0.0, -0.0]; the reciprocal of negative zero gives [-Inf, -Inf, -Inf, -Inf]; the reciprocal of positive zero gives [+Inf, +Inf, +Inf, +Inf]; and the reciprocal of positive infinity gives [0.0, 0.0, 0.0, 0.0].

```
    t.x = source0.c;
    if (negate0) {
      t.x = -t.x;
    }
    if (t.x == 1.0f) {
      u.x = 1.0f;
    } else {
      u.x = 1.0f / t.x;
    }
    if (xmask) destination.x = u.x;
    if (ymask) destination.y = u.x;
    if (zmask) destination.z = u.x;
    if (wmask) destination.w = u.x;
```

where

```
    | u.x - IEEE(1.0f/t.x) | < 1.0f/(2^22)
```

for 1.0f <= t.x <= 2.0f. The intent of this precision requirement is that this amount of relative precision apply over all values of t.x.

2.14.1.10.7 RSQ: Reciprocal Square Root

The RSQ instruction assigns the inverse square root of the absolute value of the source scalar into the destination register.

Additionally, RSQ(0.0) gives [+Inf, +Inf, +Inf, +Inf]; and both RSQ(+Inf) and RSQ(-Inf) give [0.0, 0.0, 0.0, 0.0];

```
    t.x = source0.c;
    if (negate0) {
      t.x = -t.x;
    }
    u.x = 1.0f / sqrt(fabs(t.x));
    if (xmask) destination.x = u.x;
    if (ymask) destination.y = u.x;
    if (zmask) destination.z = u.x;
    if (wmask) destination.w = u.x;
```

where

```
    | u.x - IEEE(1.0f/sqrt(fabs(t.x))) | < 1.0f/(2^22)
```

for 1.0f <= t.x <= 4.0f. The intent of this precision requirement is that this amount of relative precision apply over all values of t.x.

2.14.1.10.8 DP3: Three-Component Dot Product

The DP3 instruction assigns the three-component dot product of the two source vectors into the destination register.

```
t.x = source0.c***;
t.y = source0.*c**;
t.z = source0.**c*;
if (negate0) {
  t.x = -t.x;
  t.y = -t.y;
  t.z = -t.z;
}
u.x = source1.c***;
u.y = source1.*c**;
u.z = source1.**c*;
if (negate1) {
  u.x = -u.x;
  u.y = -u.y;
  u.z = -u.z;
}
v.x = t.x * u.x + t.y * u.y + t.z * u.z;
if (xmask) destination.x = v.x;
if (ymask) destination.y = v.x;
if (zmask) destination.z = v.x;
if (wmask) destination.w = v.x;
```

2.14.1.10.9 DP4: Four-Component Dot Product

The DP4 instruction assigns the four-component dot product of the two source vectors into the destination register.

```
t.x = source0.c***;
t.y = source0.*c**;
t.z = source0.**c*;
t.w = source0.***c;
if (negate0) {
  t.x = -t.x;
  t.y = -t.y;
  t.z = -t.z;
  t.w = -t.w;
}
u.x = source1.c***;
u.y = source1.*c**;
u.z = source1.**c*;
u.w = source1.***c;
if (negate1) {
  u.x = -u.x;
  u.y = -u.y;
  u.z = -u.z;
  u.w = -u.w;
}
v.x = t.x * u.x + t.y * u.y + t.z * u.z + t.w * u.w;
if (xmask) destination.x = v.x;
if (ymask) destination.y = v.x;
if (zmask) destination.z = v.x;
if (wmask) destination.w = v.x;
```

2.14.1.10.10 DST: Distance Vector

The DST instructions calculates a distance vector for the values of two source vectors. The first vector is assumed to be [NA, d*d, d*d, NA] and the second source vector is assumed to be [NA, 1.0/d, NA, 1.0/d], where the value of a component labeled NA is undefined. The destination vector is then assigned [1,d,d*d,1.0/d].

```
    t.y = source0.*c**;
    t.z = source0.**c*;
    if (negate0) {
      t.y = -t.y;
      t.z = -t.z;
    }
    u.y = source1.*c**;
    u.w = source1.***c;
    if (negate1) {
      u.y = -u.y;
      u.w = -u.w;
    }
    if (xmask) destination.x = 1.0;
    if (ymask) destination.y = t.y*u.y;
    if (zmask) destination.z = t.z;
    if (wmask) destination.w = u.w;
```

2.14.1.10.11 MIN: Minimum

The MIN instruction assigns the component-wise minimum of the two source vectors into the destination register.

```
    t.x = source0.c***;
    t.y = source0.*c**;
    t.z = source0.**c*;
    t.w = source0.***c;
    if (negate0) {
      t.x = -t.x;
      t.y = -t.y;
      t.z = -t.z;
      t.w = -t.w;
    }
    u.x = source1.c***;
    u.y = source1.*c**;
    u.z = source1.**c*;
    u.w = source1.***c;
    if (negate1) {
      u.x = -u.x;
      u.y = -u.y;
      u.z = -u.z;
      u.w = -u.w;
    }
    if (xmask) destination.x = (t.x < u.x) ? t.x : u.x;
    if (ymask) destination.y = (t.y < u.y) ? t.y : u.y;
    if (zmask) destination.z = (t.z < u.z) ? t.z : u.z;
    if (wmask) destination.w = (t.w < u.w) ? t.w : u.w;
```

2.14.1.10.12 MAX: Maximum

The MAX instruction assigns the component-wise maximum of the two source vectors into the destination register.

```
    t.x = source0.c***;
    t.y = source0.*c**;
    t.z = source0.**c*;
    t.w = source0.***c;
    if (negate0) {
      t.x = -t.x;
      t.y = -t.y;
      t.z = -t.z;
      t.w = -t.w;
    }
    u.x = source1.c***;
    u.y = source1.*c**;
    u.z = source1.**c*;
    u.w = source1.***c;
    if (negate1) {
      u.x = -u.x;
      u.y = -u.y;
      u.z = -u.z;
      u.w = -u.w;
    }
    if (xmask) destination.x = (t.x >= u.x) ? t.x : u.x;
    if (ymask) destination.y = (t.y >= u.y) ? t.y : u.y;
    if (zmask) destination.z = (t.z >= u.z) ? t.z : u.z;
    if (wmask) destination.w = (t.w >= u.w) ? t.w : u.w;
```

2.14.1.10.13 SLT: Set On Less Than

The SLT instruction performs a component-wise assignment of either 1.0 or 0.0 into the destination register. 1.0 is assigned if the value of the first source vector is less than the value of the second source vector; otherwise, 0.0 is assigned.

```
    t.x = source0.c***;
    t.y = source0.*c**;
    t.z = source0.**c*;
    t.w = source0.***c;
    if (negate0) {
      t.x = -t.x;
      t.y = -t.y;
      t.z = -t.z;
      t.w = -t.w;
    }
    u.x = source1.c***;
    u.y = source1.*c**;
    u.z = source1.**c*;
    u.w = source1.***c;
    if (negate1) {
      u.x = -u.x;
      u.y = -u.y;
      u.z = -u.z;
      u.w = -u.w;
    }
    if (xmask) destination.x = (t.x < u.x) ? 1.0 : 0.0;
    if (ymask) destination.y = (t.y < u.y) ? 1.0 : 0.0;
    if (zmask) destination.z = (t.z < u.z) ? 1.0 : 0.0;
    if (wmask) destination.w = (t.w < u.w) ? 1.0 : 0.0;
```

2.14.1.10.14 SGE: Set On Greater or Equal Than

The SGE instruction performs a component-wise assignment of either 1.0 or 0.0 into the destination register. 1.0 is assigned if the value of the first source vector is greater than or equal the value of the second source vector; otherwise, 0.0 is assigned.

```
    t.x = source0.c***;
    t.y = source0.*c**;
    t.z = source0.**c*;
    t.w = source0.***c;
    if (negate0) {
      t.x = -t.x;
      t.y = -t.y;
      t.z = -t.z;
      t.w = -t.w;
    }
    u.x = source1.c***;
    u.y = source1.*c**;
    u.z = source1.**c*;
    u.w = source1.***c;
    if (negate1) {
      u.x = -u.x;
      u.y = -u.y;
      u.z = -u.z;
      u.w = -u.w;
    }
    if (xmask) destination.x = (t.x >= u.x) ? 1.0 : 0.0;
    if (ymask) destination.y = (t.y >= u.y) ? 1.0 : 0.0;
    if (zmask) destination.z = (t.z >= u.z) ? 1.0 : 0.0;
    if (wmask) destination.w = (t.w >= u.w) ? 1.0 : 0.0;
```

2.14.1.10.15 EXP: Exponential Base 2

The EXP instruction generates an approximation of the exponential base 2 for the value of a source scalar. This approximation is assigned to the z component of the destination register. Additionally, the x and y components of the destination register are assigned values useful for determining a more accurate approximation. The exponential base 2 of the source scalar can be better approximated by destination.x*FUNC(destination.y) where FUNC is some user approximation (presumably implemented by subsequent instructions in the vertex program) to 2^destination.y where 0.0 <= destination.y < 1.0.

Additionally, EXP(-Inf) or if the exponential result underflows gives [0.0, 0.0, 0.0, 1.0]; and EXP(+Inf) or if the exponential result overflows gives [+Inf, 0.0, +Inf, 1.0].

```
    t.x = source0.c;
    if (negate0) {
      t.x = -t.x;
    }
    q.x = 2^floor(t.x);
    q.y = t.x - floor(t.x);
    q.z = q.x * APPX(q.y);
    if (xmask) destination.x = q.x;
    if (ymask) destination.y = q.y;
    if (zmask) destination.z = q.z;
    if (wmask) destination.w = 1.0;
```

where APPX is an implementation dependent approximation of exponential base 2 such that

```
    | exp(q.y*log(2.0))-APPX(q.y) | < 1/(2^11)
```

for all 0 <= q.y < 1.0.

The expression "2^floor(t.x)" should overflow to +Inf and underflow to zero.

2.14.1.10.16 LOG: Logarithm Base 2

The LOG instruction generates an approximation of the logarithm base 2 for the absolute value of a source scalar. This approximation is assigned to the z component of the destination register. Additionally, the x and y components of the destination register are assigned values useful for determining a more accurate approximation. The logarithm base 2 of the absolute value of the source scalar can be better approximated by destination.x+FUNC(destination.y) where FUNC is some user approximation (presumably implemented by subsequent instructions in the vertex program) of log2(destination.y) where 1.0 <= destination.y < 2.0.

Additionally, LOG(0.0) gives [-Inf, 1.0, -Inf, 1.0]; and both LOG(+Inf) and LOG(-Inf) give [+Inf, 1.0, +Inf, 1.0].

```
    t.x = source0.c;
    if (negate0) {
      t.x = -t.x;
    }
    if (fabs(t.x) != 0.0f) {
      if (fabs(t.x) == +Inf) {
        q.x = +Inf;
        q.y = 1.0;
        q.z = +Inf;
      } else {
        q.x = Exponent(t.x);
        q.y = Mantissa(t.x);
        q.z = q.x + APPX(q.y);
      }
    } else {
      q.x = -Inf;
      q.y = 1.0;
      q.z = -Inf;
    }
    if (xmask) destination.x = q.x;
    if (ymask) destination.y = q.y;
    if (zmask) destination.z = q.z;
    if (wmask) destination.w = 1.0;
```

where APPX is an implementation dependent approximation of logarithm base 2 such that

```
    | log(q.y)/log(2.0) - APPX(q.y) | < 1/(2^11)
```

for all 1.0 <= q.y < 2.0.

The "Exponent(t.x)" function returns the unbiased exponent between -126 and 127. For example, "Exponent(1.0)" equals 0.0. (Note that the IEEE floating-point representation maintains the exponent as a biased value.) Larger or smaller exponents should generate +Inf or -Inf respectively. The "Mantissa(t.x)" function returns a value in the range [1.0f, 2.0). The intent of these functions is that fabs(t.x) is approximately "Mantissa(t.x)*2^Exponent(t.x)".

2.14.1.10.17 LIT: Light Coefficients

The LIT instruction is intended to compute ambient, diffuse, and specular lighting coefficients from a diffuse dot product, a specular dot product, and a specular power that is clamped to (-128,128) exclusive. The x component of the source vector is assumed to contain a diffuse dot product (unit normal vector dotted with a unit light vector). The y component of the source vector is assumed to contain a Blinn specular dot product (unit normal vector dotted with a unit half-angle vector). The w component is assumed to contain a specular power.

An implementation must support at least 8 fraction bits in the specular power. Note that because 0.0 times anything must be 0.0, taking any base to the power of 0.0 will yield 1.0.

```
t.x = source0.c***;
t.y = source0.*c**;
t.w = source0.***c;
if (negate0) {
  t.x = -t.x;
  t.y = -t.y;
  t.w = -t.w;
}
if (t.w < -(128.0-epsilon)) t.w = -(128.0-epsilon);
else if (t.w > 128-epsilon) t.w = 128-epsilon;
if (t.x < 0.0) t.x = 0.0;
if (t.y < 0.0) t.y = 0.0;
if (xmask) destination.x = 1.0;
if (ymask) destination.y = t.x;
if (zmask) destination.z = (t.x > 0.0) ? EXP(t.w*LOG(t.y)) : 0.0;
if (wmask) destination.w = 1.0;
```

where EXP and LOG are functions that approximate the exponential base 2 and logarithm base 2 with the identical accuracy and special case requirements of the EXP and LOG instructions. epsilon is 1.0/256.0 or approximately 0.0039 which would correspond to representing the specular power with a s8.8 representation.

2.14.1.11 Vertex Program Floating Point Requirements

All vertex program calculations are assumed to use IEEE single precision floating-point math with a format of s1e8m23 (one signed bit, 8 bits of exponent, 23 bits of magnitude) or better and the round-to-zero rounding mode. The only exceptions to this are the RCP, RSQ, LOG, EXP, and LIT instructions.

Note that (positive or negative) 0.0 times anything is (positive) 0.0.

The RCP and RSQ instructions deliver results accurate to 1.0/(2^22) and the approximate output (the z component) of the EXP and LOG instructions only has to be accurate to 1.0/(2^11). The LIT instruction specular output (the z component) is allowed an error equivalent to the combination of the EXP and LOG combination to implement a power function.

The floor operations used by the ARL and EXP instructions must operate identically. Specifically, the EXP instruction's floor(t.x) intermediate result must exactly match the integer stored in the address register by the ARL instruction.

Since distance is calculated as (d^2)*(1/sqrt(d^2)), 0.0 multiplied by anything must be 0.0. This affects the MUL, MAD, DP3, DP4, DST, and LIT instructions.

Because if/then/else conditional evaluation is done by multiplying by 1.0 or 0.0 and adding, the floating point computations require:

```
  0.0 * x = 0.0    for all x (including +Inf, -Inf, +NaN, and -NaN)
  1.0 * x = x      for all x (including +Inf and -Inf)
  0.0 + x = x      for all x (including +Inf and -Inf)
```

Including +Inf, -Inf, +NaN, and -NaN when applying the above three rules is recommended but not required. (The recommended inclusion of +Inf, -Inf, +NaN, and -NaN when applying the first rule is inconsistent with IEEE floating-point requirements.)

For the purpose of comparisons performed by the SGE and SLT instructions, -0.0 is less than +0.0, -NaN is less than -Inf, and +NaN is greater than +Inf. (This is inconsistent with IEEE floating-point requirements).

No floating-point exceptions or interrupts are generated. Denorms are not supported; if a denorm is input, it is treated as 0.0 (ie, denorms are flushed to zero).

Computations involving +NaN or -NaN generate +NaN, except for the requirement that zero times +NaN or -NaN must always be zero. (This exception is inconsistent with IEEE floating-point requirements).

2.14.2 Vertex Program Update for the Current Raster Position

When vertex programs are enabled, the raster position is determined by the current vertex program. The raster position specified by RasterPos is treated as if they were specified in a Vertex command. The contents of vertex result register set is used to update respective raster position state.

Assuming an existent program, the homogeneous clip-space coordinates are passed to clipping as if they represented a point and assuming no client-defined clip planes are enabled. If the point is not culled, then the projection to window coordinates is computed (section 2.10) and saved as the current raster position and the valid bit is set. If the current vertex program is nonexistent or the "point" is culled, the current raster position and its associated data become indeterminate and the raster position valid bit is cleared.

2.14.3 Vertex Arrays for Vertex Attributes

Data for vertex attributes in vertex program mode may be specified using vertex array commands. The client may specify and enable any of sixteen vertex attribute arrays.

The vertex attribute arrays are ignored when vertex program mode is disabled. When vertex program mode is enabled, vertex attribute arrays are used.

The command

```
  void VertexAttribPointerNV(uint index, int size, enum type,
                             sizei stride, const void *pointer);
```

describes the locations and organizations of the sixteen vertex attribute arrays. index specifies the particular vertex attribute to be described. size indicates the number of values per vertex that are stored in the array; size must be one of 1, 2, 3, or 4. type specifies the data type of the values stored in the array. type must be one of SHORT, FLOAT, DOUBLE, or UNSIGNED_BYTE and these values correspond to the array types short, int, float, double, and ubyte respectively. The INVALID_OPERATION error is generated if type is UNSIGNED_BYTE and size is not 4. The INVALID_VALUE error is generated if index is greater than 15. The INVALID_VALUE error is generated if stride is negative.

The one, two, three, or four values in an array that correspond to a single vertex attribute comprise an array element. The values within each array element at stored sequentially in memory. If the stride is specified as zero, then array elements are stored sequentially as well. Otherwise points to the ith and (i+1)st elements of an array differ by stride basic machine units (typically unsigned bytes), the pointer to the (i+1)st element being greater. pointer specifies the location in memory of the first value of the first element of the array being specified.

Vertex attribute arrays are enabled with the EnableClientState command and disabled with the DisableClientState command. The value of the argument to either command is VERTEX_ATTRIB_ARRAYi_NV where i is an integer between 0 and 15; specifying a value of i enables or disables the vertex attribute array with index i. The constants obey VERTEX_ATTRIB_ARRAYi_NV = VERTEX_ATTRIB_ARRAY0_NV + i.

When vertex program mode is enabled, the ArrayElement command operates as described in this section in contrast to the behavior described in section 2.8. Likewise, any vertex array transfer commands that are defined in terms of ArrayElement (DrawArrays, DrawElements, and DrawRangeElements) assume the operation of ArrayElement described in this section when vertex program mode is enabled.

When vertex program mode is enabled, the ArrayElement command transfers the ith element of particular enabled vertex arrays as described below. For each enabled vertex attribute array, it is as though the corresponding command from section 2.14.1.1 were called with a pointer to element i. For each vertex attribute, the corresponding command is VertexAttrib[size][type]v, where size is one of [1,2,3,4], and type is one of [s,f,d,ub], corresponding to the array types short, int, float, double, and ubyte respectively.

However, if a given vertex attribute array is disabled, but its corresponding aliased conventional per-vertex parameter's vertex array (as described in section 2.14.1.6) is enabled, then it is as though the corresponding command from section 2.7 or section 2.6.2 were called with a pointer to element i. In this case, the corresponding command is determined as described in section 2.8's description of ArrayElement.

If the vertex attribute array 0 is enabled, it is as though VertexAttrib[size][type]v(0, ...) is executed last, after the executions of other corresponding commands. If the vertex attribute array 0 is disabled but the vertex array is enabled, it is as though Vertex[size][type]v is executed last, after the executions of other corresponding commands.

2.14.4 Vertex State Programs

Vertex state programs share the same instruction set as and a similar execution model to vertex programs. While vertex program are executed implicitly when a vertex transformation is provoked, vertex state programs are executed explicitly, independently of any vertices. Vertex state programs can write program parameter registers, but may not write vertex result registers.

The purpose of a vertex state program is to update program parameter registers by means of an application-defined program. Typically, an application will load a set of program parameters and then execute a vertex state program that reads and updates the program parameter registers. For example, a vertex state program might normalize a set of unnormalized vectors previously loaded as program parameters. The expectation is that subsequently executed vertex programs would use the normalized program parameters.

Vertex state programs are loaded with the same LoadProgramNV command (see section 2.14.1.7) used to load vertex programs except that the target must be VERTEX_STATE_PROGRAM_NV when loading a vertex state program.

Vertex state programs must conform to a more limited grammar than the grammar for vertex programs. The vertex state program grammar for syntactically valid sequences is the same as the grammar defined in section 2.14.1.7 with the following modified rules:

```
<program>              ::= "!!VSP1.0" <instructionSequence> "END"

<dstReg>               ::= <absProgParamReg>
                         | <temporaryReg>

<vertexAttribReg>      ::= "v" "[" "0" "]"
```

A vertex state program fails to load if it does not write at least one program parameter register.

A vertex state program fails to load if it contains more than 128 instructions.

A vertex state program fails to load if any instruction sources more than one unique program parameter register.

A vertex state program fails to load if any instruction sources more than one unique vertex attribute register (this is necessarily true because only vertex attribute 0 is available in vertex state programs).

The error INVALID_OPERATION is generated if a vertex state program fails to load because it is not syntactically correct or for one of the other reasons listed above.

A successfully loaded vertex state program is parsed into a sequence of instructions. Each instruction is identified by its tokenized name. The operation of these instructions when executed is defined in section 2.14.1.10.

Executing vertex state programs is legal only outside a Begin/End pair. A vertex state program may not read any vertex attribute register other than register zero. A vertex state program may not write any vertex result register.

The command

```
  ExecuteProgramNV(enum target, uint id, const float *params);
```

executes the vertex state program named by id. The target must be VERTEX_STATE_PROGRAM_NV and the id must be the name of program loaded with a target type of VERTEX_STATE_PROGRAM_NV. params points to an array of four floating-point values that are loaded into vertex attribute register zero (the only vertex attribute readable from a vertex state program).

The INVALID_OPERATION error is generated if the named program is nonexistent, is invalid, or the program is not a vertex state program. A vertex state program may not be valid for reasons explained in section 2.14.5.

2.14.5 Tracking Matrices

As a convenience to applications, standard GL matrix state can be tracked into program parameter vectors. This permits vertex programs to access matrices specified through GL matrix commands.

In addition to GL's conventional matrices, several additional matrices are available for tracking. These matrices have names of the form MATRIXi_NV where i is between zero and n-1 where n is the value of the MAX_TRACK_MATRICES_NV implementation dependent constant. The MATRIXi_NV constants obey MATRIXi_NV = MATRIX0_NV + i. The value of MAX_TRACK_MATRICES_NV must be at least eight. The maximum stack depth for tracking matrices is defined by the MAX_TRACK_MATRIX_STACK_DEPTH_NV and must be at least 1.

The command

```
  TrackMatrixNV(enum target, uint address, enum matrix, enum transform);
```

tracks a given transformed version of a particular matrix into a contiguous sequence of four vertex program parameter registers beginning at address. target must be VERTEX_PROGRAM_NV (though tracked matrices apply to vertex state programs as well because both vertex state programs and vertex programs shared the same program parameter registers). matrix must be one of NONE, MODELVIEW, PROJECTION, TEXTURE, TEXTUREi_ARB (where i is between 0 and n-1 where n is the number of texture units supported), COLOR (if the ARB_imaging subset is supported), MODELVIEW_PROJECTION_NV, or MATRIXi_NV. transform must be one of IDENTITY_NV, INVERSE_NV, TRANSPOSE_NV, or INVERSE_TRANSPOSE_NV. The INVALID_VALUE error is generated if address is not a multiple of four.

The MODELVIEW_PROJECTION_NV matrix represents the concatenation of the current modelview and projection matrices. If M is the current modelview matrix and P is the current projection matrix, then the MODELVIEW_PROJECTION_NV matrix is C and computed as

```
    C = P M
```

Matrix tracking for the specified program parameter register and the next consecutive three registers is disabled when NONE is supplied for matrix. When tracking is disabled the previously tracked program parameter registers retain the state of their last tracked values. Otherwise, the specified transformed version of matrix is tracked into the specified program parameter register and the next three registers. Whenever the matrix changes, the transformed version of the matrix is updated in the specified range of program parameter registers. If TEXTURE is specified for matrix, the texture matrix for the current active texture unit is tracked. If TEXTUREi_ARB is specified for matrix, the <i>th texture matrix is tracked.

Matrices are tracked row-wise meaning that the top row of the transformed matrix is loaded into the program parameter address, the second from the top row of the transformed matrix is loaded into the program parameter address+1, the third from the top row of the transformed matrix is loaded into the program parameter address+2, and the bottom row of the transformed matrix is loaded into the program parameter address+3. The transformed matrix may be identical to the specified matrix, the inverse of the specified matrix, the transpose of the specified matrix, or the inverse transpose of the specified matrix, depending on the value of transform.

When matrix tracking is enabled for a particular program parameter register sequence, updates to the program parameter using ProgramParameterNV commands, a vertex program, or a vertex state program are not possible. The INVALID_OPERATION error is generated if a ProgramParameterNV command is used to update a program parameter register currently tracking a matrix.

The INVALID_OPERATION error is generated by ExecuteProgramNV when the vertex state program requested for execution writes to a program parameter register that is currently tracking a matrix because the program is considered invalid.

2.14.6 Required Vertex Program State

The state required for vertex programs consists of:

  a bit indicating whether or not program mode is enabled;

a bit indicating whether or not two-sided color mode is enabled;

a bit indicating whether or not program-specified point size mode is enabled;

96 4-component floating-point program parameter registers;

16 4-component vertex attribute registers (though this state is aliased with the current normal, primary color, secondary color, fog coordinate, weights, and texture coordinate sets);

24 sets of matrix tracking state for each set of four sequential program parameter registers, consisting of a n-valued integer indicated the tracked matrix or GL_NONE (where n is 5 + the number of texture units supported + the number of tracking matrices supported) and a four-valued integer indicating the transformation of the tracked matrix;

an unsigned integer naming the currently bound vertex program

and the state must be maintained to indicate which integers are currently in use as program names.

Each existent program object consists of a target, a boolean indicating whether the program is resident, an array of type ubyte containing the program string, and the length of the program string array. Initially, no program objects exist.

Program mode, two-sided color mode, and program-specified point size mode are all initially disabled.

The initial state of all 96 program parameter registers is (0,0,0,0).

The initial state of the 16 vertex attribute registers is (0,0,0,1) except in cases where a vertex attribute register aliases to a conventional GL transform mode vertex parameter in which case the initial state is the initial state of the respective aliased conventional vertex parameter.

The initial state of the 24 sets of matrix tracking state is NONE for the tracked matrix and IDENTITY_NV for the transformation of the tracked matrix.

The initial currently bound program is zero.

The client state required to implement the 16 vertex attribute arrays consists of 16 boolean values, 16 memory pointers, 16 integer stride values, 16 symbolic constants representing array types, and 16 integers representing values per element. Initially, the boolean values are each disabled, the memory pointers are each null, the strides are each zero, the array types are each FLOAT, and the integers representing values per element are each four."

Additions to Chapter 3 of the OpenGL 1.2.1 Specification (Rasterization)

-- Section 3.3 "Points"

Change the first paragraph to read:

"When program vertex mode is disabled, the point size for rasterizing points is controlled with

```
void PointSize(float size);
```

size specifies the width or diameter of a point. The initial point size value is 1.0. A value less than or equal to zero results in the error INVALID_VALUE. When vertex program mode is enabled, the point size for rasterizing points is determined as described in section 2.14.1.5."

-- Section 3.9 "Color Sum"

Change the first paragraph to read:

"At the beginning of color sum, a fragment has two RGBA colors: a primary color cpri (which texturing, if enabled, may have modified) and a secondary color csec. If vertex program mode is disabled, csec is defined by the lighting equations in section 2.13.1. If vertex program mode is enabled, csec is the fragment's secondary color, obtained by interpolating the COL1 (or BFC1 if the primitive is a polygon, the vertex program two-sided color mode is enabled, and the polygon is back-facing) vertex result register RGB components for the vertices making up the primitive; the alpha component of csec when program mode is enabled is always zero. The components of these two colors are summed to produce a single post-texturing RGBA color c. The components of c are then clamped to the range [0,1]."

-- Section 3.10 "Fog"

Change the initial sentences in the second paragraph to read:

"This factor f may be computed according to one of three equations:

```
        f = exp(-d*c)                              (3.24)
        f = exp(-(d*c)^2)                          (3.25)
        f = (e-c)/(e-s)                            (3.26)
```

If vertex program mode is enabled, then c is the fragment's fog coordinate, obtained by interpolating the FOGC vertex result register values for the vertices making up the primitive. When vertex program mode is disabled, the c is the eye-coordinate distance from the eye, (0,0,0,1) in eye-coordinates, to the fragment center." ...

Additions to Chapter 4 of the OpenGL 1.2.1 Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the OpenGL 1.2.1 Specification (Special Functions)

-- Section 5.1 "Evaluators"

Add the following lines to the end of table 5.1 (page 165):

target	k	values
MAP1_VERTEX_ATTRIB0_4_NV	4	x, y, z, w vertex attribute 0
MAP1_VERTEX_ATTRIB1_4_NV	4	x, y, z, w vertex attribute 1
MAP1_VERTEX_ATTRIB2_4_NV	4	x, y, z, w vertex attribute 2
MAP1_VERTEX_ATTRIB3_4_NV	4	x, y, z, w vertex attribute 3
MAP1_VERTEX_ATTRIB4_4_NV	4	x, y, z, w vertex attribute 4
MAP1_VERTEX_ATTRIB5_4_NV	4	x, y, z, w vertex attribute 5
MAP1_VERTEX_ATTRIB6_4_NV	4	x, y, z, w vertex attribute 6
MAP1_VERTEX_ATTRIB7_4_NV	4	x, y, z, w vertex attribute 7
MAP1_VERTEX_ATTRIB8_4_NV	4	x, y, z, w vertex attribute 8
MAP1_VERTEX_ATTRIB9_4_NV	4	x, y, z, w vertex attribute 9
MAP1_VERTEX_ATTRIB10_4_NV	4	x, y, z, w vertex attribute 10
MAP1_VERTEX_ATTRIB11_4_NV	4	x, y, z, w vertex attribute 11
MAP1_VERTEX_ATTRIB12_4_NV	4	x, y, z, w vertex attribute 12
MAP1_VERTEX_ATTRIB13_4_NV	4	x, y, z, w vertex attribute 13
MAP1_VERTEX_ATTRIB14_4_NV	4	x, y, z, w vertex attribute 14
MAP1_VERTEX_ATTRIB15_4_NV	4	x, y, z, w vertex attribute 15

Replace the four paragraphs on pages 167-168 that explain the operation of EvalCoord:

"EvalCoord operates differently depending on whether vertex program mode is enabled or not. We first discuss how EvalCoord operates when vertex program mode is disabled.

When one of the EvalCoord commands is issued and vertex program mode is disabled, all currently enabled maps (excluding the maps that correspond to vertex attributes, i.e. maps of the form MAPx_VERTEX_ATTRIBn_4_NV). ..."

Add a paragraph before the initial paragraph discussing AUTO_NORMAL:

"When one of the EvalCoord commands is issued and vertex program mode is enabled, the evaluation and the issuing of per-vertex parameter commands matches the discussion above, except that if any vertex attribute maps are enabled, the corresponding VertexAttribNV call for each enabled vertex attribute map is issued with the map's evaluated coordinates and the corresponding aliased per-vertex parameter map is ignored if it is also enabled, with one important difference. As is the case when vertex program mode is disabled, the GL uses evaluated values instead of current values for those evaluations that are enabled (otherwise the current values are used). The order of the effective commands is immaterial, except that Vertex or VertexAttribNV(0, ...) (the commands that issue provoke vertex program execution) must be issued last. Use of evaluators has no effect on the current vertex attributes or conventional per-vertex parameters. If a vertex attribute map is disabled, but its corresponding conventional per-vertex parameter map is enabled, the conventional per-vertex parameter map is evaluated and issued as when vertex program mode is not enabled."

Replace the two paragraphs discussing AUTO_NORMAL with:

"Finally, if either MAP2_VERTEX_3 or MAP2_VERTEX_4 is enabled or if both MAP2_VERTEX_ATTRIB0_4_NV and vertex program mode are enabled, then the normal to the surface is computed. Analytic computation, which sometimes yields normals of length zero, is one method which may be used. If automatic normal generation is enabled, then this computed normal is used as the normal associated with a generated vertex (when program mode is disabled) or as vertex attribute 2 (when vertex program mode is enabled). Automatic normal generation is controlled with Enable and Disable with the symbolic constant AUTO_NORMAL. If automatic normal generation is disabled and vertex program mode is enabled, then vertex attribute 2 is evaluated as usual. If automatic normal generation and vertex program mode are disabled, then a corresponding normal map, if enabled, is used to produce a normal. If neither automatic normal generation nor a map corresponding to the normal per-vertex parameter (or vertex attribute 2 in program mode) are enabled, then no normal is sent with a vertex resulting from an evaluation (the effect is that the current normal is used). For MAP_VERTEX3, let q=p. For MAP_VERTEX_4 or MAP2_VERTEX_ATTRBI0_4_NV, let q = (x/w, y/w, z/w) where (x,y,z,w)=p. Then let

```
   m = (partial q / partial u) cross (partial q / partial v)
```

Then when vertex program mode is disabled, the generated analytic normal, n, is given by n=m/||m||. However, when vertex program mode is enabled, the generated analytic normal used for vertex attribute 2 is simply (mx,my,mz,1). In vertex program mode, the normalization of the generated analytic normal can be performed by the current vertex program."

Change the respective sentences of the last paragraph discussing required evaluator state to read:

"The state required for evaluators potentially consists of 9 conventional one-dimensional map specifications, 16 vertex attribute one-dimensional map specifications, 9 conventional two-dimensional map specifications, and 16 vertex attribute two-dimensional map specifications indicating which are enabled. ... All vertex coordinate maps produce the coordinates (0,0,0,1) (or the appropriate subset); all normal coordinate maps produce (0,0,1); RGBA maps produce (1,1,1,1); color index maps produce 1.0; texture coordinate maps produce (0,0,0,1); and vertex attribute maps produce (0,0,0,1). ... If any evaluation command is issued when none of MAPn_VERTEX_3, MAPn_VERTEX_4, or MAPn_VERTEX_ATTRIB0_NV (where n is the map dimension being evaluated) are enabled, nothing happens."

-- Section 5.4 "Display Lists"

Add to the list of commands not compiled into display lists in the third to the last paragraph:

"AreProgramsResidentNV, IsProgramNV, GenProgramsNV, DeleteProgramsNV, VertexAttribPointerNV"

Additions to Chapter 6 of the OpenGL 1.2.1 Specification (State and State Requests)

-- Section 6.1.12 "Saving and Restoring State"

Only the enables and vertex array state introduced by this extension can be pushed and popped.

See the attribute column in table X.5 for determining what vertex program state can be pushed and popped with PushAttrib, PopAttrib, PushClientAttrib, and PopClientAttrib.

The new evaluator enables in table 6.22 can also be pushed and popped.

-- NEW Section 6.1.13 "Vertex Program Queries"

"The commands

```
  void GetProgramParameterfvNV(enum target, uint index,
                               enum pname, float *params);
  void GetProgramParameterdvNV(enum target, uint index,
                               enum pname, double *params);
```

obtain the current program parameters for the given program target and parameter index into the array params. target must be VERTEX_PROGRAM_NV. pname must be PROGRAM_PARAMETER_NV. The INVALID_VALUE error is generated if index is greater than 95. Each program parameter is an array of four values.

The command

```
  void GetProgramivNV(uint id, enum pname, int *params);
```

obtains program state named by pname for the program named id in the array params. pname must be one of PROGRAM_TARGET_NV, PROGRAM_LENGTH_NV, or PROGRAM_RESIDENT_NV. The INVALID_OPERATION error is generated if the program named id does not exist.

The command

```
  void GetProgramStringNV(uint id, enum pname,
                          ubyte *program);
```

obtains the program string for program id. pname must be PROGRAM_STRING_NV. n ubytes are returned into the array program where n is the length of the program in ubytes. GetProgramivNV with PROGRAM_LENGTH_NV can be used to query the length of a program's string. The INVALID_OPERATION error is generated if the program named id does not exist.

The command

```
  void GetTrackMatrixivNV(enum target, uint address,
                          enum pname, int *params);
```

obtains the matrix tracking state named by pname for the specified address in the array params. target must be VERTEX_PROGRAM_NV. pname must be either TRACK_MATRIX_NV or TRACK_MATRIX_TRANSFORM_NV. If the matrix tracked is a texture matrix, TEXTUREi_ARB is returned (never TEXTURE) where i indicates the texture unit of the particular tracked texture matrix. The INVALID_VALUE error is generated if address is not divisible by four and is not less than 96.

The commands

```
  void GetVertexAttribdvNV(uint index, enum pname, double *params);
  void GetVertexAttribfvNV(uint index, enum pname, float *params);
  void GetVertexAttribivNV(uint index, enum pname, int *params);
```

obtain the vertex attribute state named by pname for the vertex attribute numbered index. pname must be one of ATTRIB_ARRAY_SIZE_NV, ATTRIB_ARRAY_STRIDE_NV, ATTRIB_ARRAY_TYPE_NV, or CURRENT_ATTRIB_NV. Note that all the queries except CURRENT_ATTRIB_NV return client state. The INVALID_VALUE error is generated if index is greater than 15, or if index is zero and pname is CURRENT_ATTRIB_NV.

The command

```
  void GetVertexAttribPointervNV(uint index,
                                 enum pname, void **pointer);
```

obtains the pointer named pname in the array params for vertex attribute numbered index. pname must be ATTRIB_ARRAY_POINTER_NV. The INVALID_VALUE error is generated if index greater than 15.

The command

```
  boolean IsProgramNV(uint id);
```

returns TRUE if program is the name of a program object. If program is zero or is a non-zero value that is not the name of a program object, or if an error condition occurs, IsProgramNV returns FALSE. A name returned by GenProgramsNV but not yet loaded with a program is not the name of a program object."

-- NEW Section 6.1.14 "Querying Current Matrix State"

"Instead of providing distinct symbolic tokens for querying each matrix and matrix stack depth, the symbolic tokens CURRENT_MATRIX_NV and CURRENT_MATRIX_STACK_DEPTH_NV in conjunction with the GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev return the respective state of the current matrix given the current matrix mode.

Querying CURRENT_MATRIX_NV and CURRENT_MATRIX_STACK_DEPTH_NV is the only means for querying the matrix and matrix stack depth of the tracking matrices described in section 2.14.5."

Additions to Appendix A of the OpenGL 1.2.1 Specification (Invariance)

Add the following rule:

"Rule X Vertex program and vertex state program instructions not relevant to the calculation of any result must have no effect on that result.

Rules X+1 Vertex program and vertex state program instructions relevant to the calculation of any result must always produce the identical result. In particular, the same instruction with the same source inputs must produce the identical result whether executed by a vertex program or a vertex state program.

Instructions relevant to the calculation of a result are any instructions in a sequence of instructions that eventually determine the source values for the calculation under consideration.

There is no guaranteed invariance between vertices transformed by conventional GL vertex transform mode and vertices transformed by vertex program mode. Multi-pass rendering algorithms that require rendering invariances to operate correctly should not mix conventional GL vertex transform mode with vertex program mode for different rendering passes. However such algorithms will operate correctly if the algorithms limit themselves to a single mode of vertex transformation."

Additions to the AGL/GLX/WGL Specifications

Program objects are shared between AGL/GLX/WGL rendering contexts if and only if the rendering contexts share display lists. No change is made to the AGL/GLX/WGL API.

Dependencies on EXT_vertex_weighting

If the EXT_vertex_weighting extension is not supported, there is no aliasing between vertex attribute 1 and the current vertex weight. Replace the contents of the last three columns in row 5 of table X.2 with dashes.

Dependencies on EXT_point_parameters

When EXT_point_parameters is supported, the amended discussion of point size determination should be further amended with the language from the EXT_point_parameters specification though the point parameters functionality only applies when vertex program mode is disabled.

Even if the EXT_point_parameters extension is not supported, the PSIZ vertex result register must operate as specified.

Dependencies on ARB_multitexture

ARB_multitexture is required to support NV_vertex_program and the value of MAX_TEXTURE_UNITS_ARB must be at least 2. If more than 8 texture units are supported, only the first 8 texture units can be assigned texture coordinates when vertex program mode is enabled. Texture units beyond 8 are implicitly disabled when vertex program mode is enabled.

Dependencies on EXT_fog_coord

If the EXT_fog_coord extension is not supported, there is no aliasing between vertex attribute 5 and the current fog coordinate. Replace the contents of the last three columns in row 5 of table X.2 with dashes.

Even if the EXT_fog_coord extension is not supported, the FOGC vertex result register must operate as specified. Note that the FOGC vertex result register behaves identically to the EXT_fog_coord extension's FOG_COORDINATE_SOURCE_EXT being FOG_COORDINATE_EXT. This means that the functionality of EXT_fog_coord is required to implement NV_vertex_program even if the EXT_fog_coord extension is not supported.

If the EXT_fog_coord extension is supported, the state of FOG_COORDINATE_SOURCE_EXT only applies when vertex program mode is disabled and the discussion in section 3.10 is further amended by the discussion of FOG_COORDINATE_SOURCE_EXT in the EXT_fog_coord specification.

Dependencies on EXT_secondary_color

If the EXT_secondary_color extension is not supported, there is no aliasing between vertex attribute 4 and the current secondary color. Replace the contents of the last three columns in row 4 of table X.2 with dashes.

Even if the EXT_secondary_color extension is not supported, the COL1 and BFC1 vertex result registers must operate as specified. These vertex result registers are required to implement OpenGL 1.2's separate specular mode within a vertex program.

GLX Protocol

Forty-five new GL commands are added.

The following thirty-five rendering commands are sent to the sever as part of a glXRender request:

```
BindProgramNV
    2           12              rendering command length
    2           4180            rendering command opcode
    4           ENUM            target
    4           CARD32          id

ExecuteProgramNV
    2           12+4*n          rendering command length
    2           4181            rendering command opcode
    4           ENUM            target
                0x8621  n=4     GL_VERTEX_STATE_PROGRAM_NV
                else    n=0     command is erroneous
    4           CARD32          id
    4*n         LISTofFLOAT32   params

RequestResidentProgramsNV
    2           8+4*n           rendering command length
    2           4182            rendering command opcode
    4           INT32           n
    n*4         CARD32          programs
```

```
        LoadProgramNV
            2           16+n+p          rendering command length
            2           4183            rendering command opcode
            4           ENUM            target
            4           CARD32          id
            4           INT32           len
            n           LISTofCARD8     n
            p                           unused, p=pad(n)

        ProgramParameter4fvNV
            2           32              rendering command length
            2           4184            rendering command opcode
            4           ENUM            target
            4           CARD32          index
            4           FLOAT32         params[0]
            4           FLOAT32         params[1]
            4           FLOAT32         params[2]
            4           FLOAT32         params[3]

        ProgramParameter4dvNV
            2           44              rendering command length
            2           4185            rendering command opcode
            4           ENUM            target
            4           CARD32          index
            8           FLOAT64         params[0]
            8           FLOAT64         params[1]
            8           FLOAT64         params[2]
            8           FLOAT64         params[3]

        ProgramParameters4fvNV
            2           16+16*n         rendering command length
            2           4186            rendering command opcode
            4           ENUM            target
            4           CARD32          index
            4           CARD32          n
            16*n        FLOAT32         params

        ProgramParameters4dvNV
            2           16+32*n         rendering command length
            2           4187            rendering command opcode
            4           ENUM            target
            4           CARD32          index
            4           CARD32          n
            32*n        FLOAT64         params

        TrackMatrixNV
            2           20              rendering command length
            2           4188            rendering command opcode
            4           ENUM            target
            4           CARD32          address
            4           ENUM            matrix
            4           ENUM            transform

        VertexAttribPointerNV is an entirely client-side command

        VertexAttrib1svNV
            2           12              rendering command length
            2           4189            rendering command opcode
            4           CARD32          index
            2           INT16           v[0]
            2                           unused
        VertexAttrib2svNV
            2           12              rendering command length
            2           4190            rendering command opcode
            4           CARD32          index
            2           INT16           v[0]
            2           INT16           v[1]

        VertexAttrib3svNV
            2           12              rendering command length
            2           4191            rendering command opcode
            4           CARD32          index
            2           INT16           v[0]
            2           INT16           v[1]
            2           INT16           v[2]
            2                           unused

        VertexAttrib4svNV
            2           12              rendering command length
            2           4192            rendering command opcode
            4           CARD32          index
            2           INT16           v[0]
            2           INT16           v[1]
            2           INT16           v[2]
            2           INT16           v[3]

        VertexAttrib1fvNV
            2           12              rendering command length
            2           4193            rendering command opcode
            4           CARD32          index
            4           FLOAT32         v[0]

        VertexAttrib2fvNV
            2           16              rendering command length
            2           4194            rendering command opcode
            4           CARD32          index
            4           FLOAT32         v[0]
            4           FLOAT32         v[1]

        VertexAttrib3fvNV
            2           20              rendering command length
            2           4195            rendering command opcode
            4           CARD32          index
            4           FLOAT32         v[0]
            4           FLOAT32         v[1]
            4           FLOAT32         v[2]

        VertexAttrib4fvNV
            2           24              rendering command length
            2           4196            rendering command opcode
            4           CARD32          index
            4           FLOAT32         v[0]
            4           FLOAT32         v[1]
            4           FLOAT32         v[2]
            4           FLOAT32         v[3]

        VertexAttrib1dvNV
            2           16              rendering command length
            2           4197            rendering command opcode
            4           CARD32          index
            8           FLOAT64         v[0]
        VertexAttrib2dvNV
            2           24              rendering command length
            2           4198            rendering command opcode
            4           CARD32          index
            8           FLOAT64         v[0]
            8           FLOAT64         v[1]

        VertexAttrib3dvNV
            2           32              rendering command length
            2           4199            rendering command opcode
            4           CARD32          index
            8           FLOAT64         v[0]
            8           FLOAT64         v[1]
            8           FLOAT64         v[2]

        VertexAttrib4dvNV
            2           40              rendering command length
            2           4200            rendering command opcode
            4           CARD32          index
            8           FLOAT64         v[0]
            8           FLOAT64         v[1]
            8           FLOAT64         v[2]
            8           FLOAT64         v[3]

        VertexAttrib4ubvNV
            2           12              rendering command length
            2           4201            rendering command opcode
            4           CARD32          index
            1           CARD8           v[0]
            1           CARD8           v[1]
            1           CARD8           v[2]
            1           CARD8           v[3]

        VertexAttribs1svNV
            2           12+2*n+p        rendering command length
            2           4202            rendering command opcode
            4           CARD32          index
            4           CARD32          n
            2*n         INT16           v
            p                           unused, p=pad(2*n)

        VertexAttribs2svNV
            2           12+4*n          rendering command length
            2           4203            rendering command opcode
            4           CARD32          index
            4           CARD32          n
            4*n         INT16           v

        VertexAttribs3svNV
            2           12+6*n+p        rendering command length
            2           4204            rendering command opcode
            4           CARD32          index
            4           CARD32          n
            6*n         INT16           v
            p                           unused, p=pad(6*n)

        VertexAttribs4svNV
            2           12+8*n          rendering command length
            2           4205            rendering command opcode
            4           CARD32          index
            4           CARD32          n
            8*n         INT16           v
        VertexAttribs1fvNV
            2           12+4*n          rendering command length
            2           4206            rendering command opcode
            4           CARD32          index
            4           CARD32          n
            4*n         FLOAT32         v

        VertexAttribs2fvNV
            2           12+8*n          rendering command length
            2           4207            rendering command opcode
            4           CARD32          index
            4           CARD32          n
            8*n         FLOAT32         v

        VertexAttribs3fvNV
            2           12+12*n         rendering command length
            2           4208            rendering command opcode
            4           CARD32          index
            4           CARD32          n
            12*n        FLOAT32         v

        VertexAttribs4fvNV
            2           12+16*n         rendering command length
            2           4209            rendering command opcode
            4           CARD32          index
            4           CARD32          n
            16*n        FLOAT32         v

        VertexAttribs1dvNV
            2           12+8*n          rendering command length
            2           4210            rendering command opcode
            4           CARD32          index
            4           CARD32          n
            8*n         FLOAT64         v

        VertexAttribs2dvNV
            2           12+16*n         rendering command length
            2           4211            rendering command opcode
            4           CARD32          index
            4           CARD32          n
            16*n        FLOAT64         v

        VertexAttribs3dvNV
            2           12+24*n         rendering command length
            2           4212            rendering command opcode
            4           CARD32          index
            4           CARD32          n
            24*n        FLOAT64         v

        VertexAttribs4dvNV
            2           12+32*n         rendering command length
            2           4213            rendering command opcode
            4           CARD32          index
            4           CARD32          n
            32*n        FLOAT64         v

        VertexAttribs4ubvNV
            2           12+4*n          rendering command length
            2           4214            rendering command opcode
            4           CARD32          index
            4           CARD32          n
            4*n         CARD8           v
```

```
    The remaining twelve commands are non-rendering commands.  These commands
    are sent separately (i.e., not as part of a glXRender or glXRenderLarge
    request), using the glXVendorPrivateWithReply request:

        AreProgramsResidentNV
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           4+n             request length
            4           1293            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           INT32           n
            n*4         LISTofCARD32    programs
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           (n+p)/4         reply length
            4           BOOL32          return value
            20                          unused
            n           LISTofBOOL      programs
            p                           unused, p=pad(n)

        DeleteProgramsNV
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           4+n             request length
            4           1294            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           INT32           n
            n*4         LISTofCARD32    programs

        GenProgramsNV
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           4               request length
            4           1295            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           INT32           n
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           n               reply length
            24                          unused
            n*4         LISTofCARD322   programs
        GetProgramParameterfvNV
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           6               request length
            4           1296            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           ENUM            target
            4           CARD32          index
            4           ENUM            pname
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           m               reply length, m=(n==1?0:n)
            4                           unused
            4           CARD32          n

            if (n=1) this follows:

            4           FLOAT32         params
            12                          unused

            otherwise this follows:

            16                          unused
            n*4         LISTofFLOAT32   params

        GetProgramParameterdvNV
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           6               request length
            4           1297            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           ENUM            target
            4           CARD32          index
            4           ENUM            pname
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           m               reply length, m=(n==1?0:n*2)
            4                           unused
            4           CARD32          n

            if (n=1) this follows:

            8           FLOAT64         params
            8                           unused

            otherwise this follows:

            16                          unused
            n*8         LISTofFLOAT64   params
```

```
GetProgramivNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           5               request length
    4           1298            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           CARD32          id
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           m               reply length, m=(n==1?0:n)
    4                           unused
    4           CARD32          n

    if (n=1) this follows:

    4           INT32           params
    12                          unused

    otherwise this follows:

    16                          unused
    n*4         LISTofINT32     params

GetProgramStringNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           5               request length
    4           1299            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           CARD32          id
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           (n+p)/4         reply length
    4                           unused
    4           CARD32          n
    16                          unused
    n           STRING          program
    p                           unused, p=pad(n)
```

```
        GetTrackMatrixivNV
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           6               request length
            4           1300            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           ENUM            target
            4           CARD32          address
            4           ENUM            pname
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           m               reply length, m=(n==1?0:n)
            4                           unused
            4           CARD32          n

            if (n=1) this follows:

            4           INT32           params
            12                          unused

            otherwise this follows:

            16                          unused
            n*4         LISTofINT32     params

        Note that ATTRIB_ARRAY_SIZE_NV, ATTRIB_ARRAY_STRIDE_NV, and
        ATTRIB_ARRAY_TYPE_NV may be queried by GetVertexAttribNV but
        return client-side state.

        GetVertexAttribdvNV
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           5               request length
            4           1301            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           INT32           index
            4           ENUM            pname
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           m               reply length, m=(n==1?0:n*2)
            4                           unused
            4           CARD32          n

            if (n=1) this follows:

            8           FLOAT64         params
            8                           unused

            otherwise this follows:

            16                          unused
            n*8         LISTofFLOAT64   params
        GetVertexAttribfvNV
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           5               request length
            4           1302            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           INT32           index
            4           ENUM            pname
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           m               reply length, m=(n==1?0:n)
            4                           unused
            4           CARD32          n

            if (n=1) this follows:

            4           FLOAT32         params
            12                          unused

            otherwise this follows:

            16                          unused
            n*4         LISTofFLOAT32   params

        GetVertexAttribivNV
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           5               request length
            4           1303            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           INT32           index
            4           ENUM            pname
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           m               reply length, m=(n==1?0:n)
            4                           unused
            4           CARD32          n

            if (n=1) this follows:

            4           INT32           params
            12                          unused

            otherwise this follows:

            16                          unused
            n*4         LISTofINT32     params

        GetVertexAttribPointervNV is an entirely client-side command
        IsProgramNV
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           4               request length
            4           1304            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           INT32           n
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           0               reply length
            4           BOOL32          return value
            20                          unused
```

Errors

The error INVALID_VALUE is generated if VertexAttribNV is called where index is greater than 15.

The error INVALID_VALUE is generated if any ProgramParameterNV has an index is greater than 95.

The error INVALID_VALUE is generated if VertexAttribPointerNV is called where index is greater than 15.

The error INVALID_VALUE is generated if VertexAttribPointerNV is called where size is not one of 1, 2, 3, or 4.

The error INVALID_VALUE is generated if VertexAttribPointerNV is called where stride is negative.

The error INVALID_OPERATION is generated if VertexAttribPointerNV is called where type is UNSIGNED_BYTE and size is not 4.

The error INVALID_VALUE is generated if LoadProgramNV is used to load a program with an id of zero.

The error INVALID_OPERATION is generated if LoadProgramNV is used to load an id that is currently loaded with a program of a different program target.

The error INVALID_OPERATION is generated if the program passed to LoadProgramNV fails to load because it is not syntactically correct based on the specified target. The value of PROGRAM_ERROR_POSITION_NV is still updated when this error is generated.

The error INVALID_OPERATION is generated if LoadProgramNV has a target of VERTEX_PROGRAM_NV and the specified program fails to load because it does not write the HPOS register at least once. The value of PROGRAM_ERROR_POSITION_NV is still updated when this error is generated.

The error INVALID_OPERATION is generated if LoadProgramNV has a target of VERTEX_STATE_PROGRAM_NV and the specified program fails to load because it does not write at least one program parameter register. The value of PROGRAM_ERROR_POSITION_NV is still updated when this error is generated.

```
    The error INVALID_OPERATION is generated if the vertex program
    or vertex state program passed to LoadProgramNV fails to load
    because it contains more than 128 instructions.  The value of
    PROGRAM_ERROR_POSITION_NV is still updated when this error is
    generated.

    The error INVALID_OPERATION is generated if a program is loaded with
    LoadProgramNV for id when id is currently loaded with a program of
    a different target.

    The error INVALID_OPERATION is generated if BindProgramNV attempts
    to bind to a program name that is not a vertex program (for example,
    if the program is a vertex state program).

    The error INVALID_VALUE is generated if GenProgramsNV is called
    where n is negative.

    The error INVALID_VALUE is generated if AreProgramsResidentNV is
    called and any of the queried programs are zero or do not exist.

    The error INVALID_OPERATION is generated if ExecuteProgramNV executes
    a program that does not exist.

    The error INVALID_OPERATION is generated if ExecuteProgramNV executes
    a program that is not a vertex state program.

    The error INVALID_OPERATION is generated if Begin, RasterPos, or a
    command that performs an explicit Begin is called when vertex program
    mode is enabled and the currently bound vertex program writes program
    parameters that are currently being tracked.

    The error INVALID_OPERATION is generated if ExecuteProgramNV is called
    and the vertex state program to execute writes program parameters
    that are currently being tracked.

    The error INVALID_VALUE is generated if TrackMatrixNV has a target
    of VERTEX_PROGRAM_NV and attempts to track an address is not a
    multiple of four.

    The error INVALID_VALUE is generated if GetProgramParameterNV is
    called to query an index greater than 95.

    The error INVALID_VALUE is generated if GetVertexAttribNV is called
    to query an index greater than 15 or equal to zero.

    The error INVALID_VALUE is generated if GetVertexAttribPointervNV
    is called to query an index greater than 15.

    The error INVALID_OPERATION is generated if GetProgramivNV is called
    and the program named id does not exist.

    The error INVALID_OPERATION is generated if GetProgramStringNV is called
    and the program named id does not exist.

    The error INVALID_VALUE is generated if GetTrackMatrixivNV is called
    with an address that is not divisible by four and not less than 96.
```

The error INVALID_VALUE is generated if AreProgramsResidentNV, DeleteProgramsNV, GenProgramsNV, or RequestResidentProgramsNV are called where n is negative.

The error INVALID_VALUE is generated if LoadProgramNV is called where len is negative.

The error INVALID_VALUE is generated if ProgramParameters4dvNV or ProgramParameters4fvNV are called where count is negative.

The error INVALID_VALUE is generated if VertexAttribs{1,2,3,4}{d,f,s}vNV is called where count is negative.

The error INVALID_ENUM is generated if BindProgramNV, GetProgramParameterfvNV, GetProgramParameterdvNV, GetTrackMatrixivNV, ProgramParameter4fNV, ProgramParameter4dNV, ProgramParameter4fvNV, ProgramParameter4dvNV, ProgramParameters4fvNV, ProgramParameters4dvNV, or TrackMatrixNV are called where <target> is not VERTEX_PROGRAM_NV.

The error INVALID_ENUM is generated if LoadProgramNV or ExecuteProgramNV are called where <target> is not either VERTEX_PROGRAM_NV or VERTEX_STATE_PROGRAM_NV.

New State

update table 6.22 (page 212) so that all the "9"s are "25"s because there are 9 conventional map targets and 16 vertex attribute map targets making a total of 25.

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
VERTEX_PROGRAM_NV	B	IsEnabled	False	vertex program enable	2.10	enable
VERTEX_PROGRAM_POINT_SIZE_NV	B	IsEnabled	False	program-specified point size mode	2.14.1.5	enable
VERTEX_PROGRAM_TWO_SIDE_NV	B	IsEnabled	False	two-sided color mode	2.14.1.5	enable
PROGRAM_ERROR_POSITION_NV	Z	GetIntegerv	-1	last program error position	2.14.1.7	-
PROGRAM_PARAMETER_NV	96xR4	GetProgramParameterNV	(0,0,0,0)	program parameters	2.14.1.2	-
CURRENT_ATTRIB_NV	16xR4	GetVertexAttribNV but zero cannot be queried, aliased with per-vertex parameters	see 2.14.6	vertex attributes	2.14.1.1	current
TRACK_MATRIX_NV	24xZ8+	GetTrackMatrixivNV	NONE	track matrix	2.14.5	-
TRACK_MATRIX_TRANSFORM_NV	24xZ8+	GetTrackMatrixivNV	IDENTITY_NV	track matrix transform	2.14.5	-
VERTEX_PROGRAM_BINDING_NV	Z+	GetIntegerv	0	bound vertex program	2.14.1.8	-
VERTEX_ATTRIB_ARRAYn_NV	16xB	IsEnabled	False	vertex attrib array enable	2.14.3	vertex-array
ATTRIB_ARRAY_SIZE_NV	16xZ	GetVertexAttribNV	4	vertex attrib array size	2.14.3	vertex-array
ATTRIB_ARRAY_STRIDE_NV	16xZ+	GetVertexAttribNV	0	vertex attrib array stride	2.14.3	vertex-array
ATTRIB_ARRAY_TYPE_NV	16xZ4	GetVertexAttribNV	FLOAT	vertex attrib array type	2.14.3	vertex-array

Table X.5. New State Introduced by NV_vertex_program.

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
PROGRAM_TARGET_NV	Z2	GetProgramivNV	0	program target	6.1.13	-
PROGRAM_LENGTH_NV	Z+	GetProgramivNV	0	program length	6.1.13	-
PROGRAM_RESIDENT_NV	Z2	GetProgramivNV	False	program residency	6.1.13	-
PROGRAM_STRING_NV	ubxn	GetProgramStringNV	""	program string	6.1.13	-

Table X.6. Program Object State.

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
-	12xR4	-	(0,0,0,0)	temporary registers	2.14.1.4	-
-	15xR4	-	(0,0,0,1)	vertex result registers	2.14.1.4	-
	Z4	-	(0,0,0,0)	vertex program address register	2.14.1.3	-

Table X.7. Vertex Program Per-vertex Execution State.

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
CURRENT_MATRIX_STACK_DEPTH_NV	m*Z+	GetIntegerv	1	current stack depth	6.1.14	-
CURRENT_MATRIX_NV	m*n*xM^4	GetFloatv	Identity	current matrix	6.1.14	-

Table X.8. Current matrix state where m is the total number of matrices including texture matrices and tracking matrices and n is the number of matrices on each particular matrix stack. Note that this state is aliased with existing matrix state.

New Implementation Dependent State

Get Value	Type	Get Command	Minimum Value	Description	Sec	Attribute
MAX_TRACK_MATRIX_STACK_DEPTH_NV	Z+	GetIntegerv	1	maximum tracking matrix stack depth	2.14.5	-
MAX_TRACK_MATRICES_NV	Z+	GetIntegerv	8 (not to exceed 32)	maximum number of tracking matrices	2.14.5	-

Table X.9. New Implementation-Dependent Values Introduced by NV_vertex_program.

Revision History

Version 1.1:

Added normalization example to Issues.

Fix explanation of EXP and ARL floor equivalence.

Clarify that vertex state programs fail if they load more than one vertex attribute (though only one is possible).

Version 1.2

Add GLX protocol for VertexAttrib4ubvNV and VertexAttribs4ubvNV

Add issue about TrackMatrixNV transform behavior with example

Fix the C code specifying VertexAttribsvNV

Version 1.3

Dropped support for INT typed vertex attrib arrays.

Clarify that when ArrayElement is executed and vertex program mode is enabled and the vertex attrib 0 array is enabled, the vertex attrib 0 array command is executed last. However when ArrayElement is executed and vertex program mode is enabled and the vertex attrib 0 array is disabled and the vertex array is enabled, the vertex array command is executed last.

Version 1.4

Allow TEXTUREi_ARB for the track matrix. This allows matrix tracking of a particular texture matrix without reference to active texture (set by glActiveTextureARB) state.

Early NVIDIA drivers (prior to October 5, 2001) have a bug in their handling of tracking matrices specified with TEXTURE. Rather than tracking the particular texture matrix indicated by the active texture state when TrackMatrixNV is called, these early drivers incorrectly track matrix the active texture's texture matrix _at track matrix validation time_. In practice this means, every tracked matrix defined with TEXTURE tracks the same matrix values; you cannot track distinct texture matrices at the same time and the texture matrix you actually track depends on the active texture matrix at validation time. This is a driver bug.

Drivers after October 5, 2001 properly track the texture matrix specified by active texture when TrackMatrix is called.

The new correct drivers can be distinguished from the old drivers at run time with the following code:

```
while (glGetError() != GL_NO_ERROR);  // Clear any pre-existing OpenGL errors.
glTrackMatrixNV(GL_VERTEX_PROGRAM_NV, 8, GL_TEXTURE0_ARB, GL_IDENTITY_NV);
if (glGetError() != GL_NO_ERROR) {
  // Old buggy pre-version 1.4 drivers with GL_TEXTURE
  // glTrackMatrixNV bug.
} else {
  // Correct new version 1.4 drivers (or later) with GL_TEXTURE
  // glTrackMatrixNV bug fixed and GL_TEXTUREi_NV support.

  // Note: you may want to untrack the matrix at this point.
}
```

Version 1.5

Earlier versions of this specification claimed for GetVertexAttribARB that it is an error to query any vertex attrib state for vertex attrib array zero. In fact, it should only be an error to query the CURRENT_ATTRIB_ARB state for vertex attrib zero; the size, stride, and type of vertex attrib array zero may be queried. Version 1.5 specifies the correct behavior.

Early NVIDIA drivers (prior to January 11, 2002) did not implement generate error when querying vertex attrib array zero state (ie, did the right thing for size, stride, and type) but not create an error when querying the current attribute values for vertex attrib array zero either.

Version 1.6

GLX opcodes and vendorpriv values assigned.

Version 1.7

Corrected matrix tracking example in the issues list to properly document row vs. column-major differences.

Version 1.8

Corrected EXP instruction; W component of result is always 1.0.

Version 1.9

Added language that for SGE and SLT, -NaN < -Inf and +NaN > +Inf.

Name

    NV_vertex_program1_1

Name Strings

    GL_NV_vertex_program1_1

Notice

IP Status

    NVIDIA Proprietary.

Status

    Version 1.0

Version

    NVIDIA Date: September 3, 2002
    $Id: //sw/main/docs/OpenGL/specs/GL_NV_vertex_program1_1.txt#7 $

Number

    266

Dependencies

Written based on the wording of the OpenGL 1.2.1 specification and requires OpenGL 1.2.1.

Assumes support for the NV_vertex_program extension.

Overview

This extension adds four new vertex program instructions (DPH, RCC, SUB, and ABS).

This extension also supports a position-invariant vertex program option. A vertex program is position-invariant when it generates the _exact_ same homogenuous position and window space position for a vertex as conventional OpenGL transformation (ignoring vertex blending and weighting).

By default, vertex programs are _not_ guaranteed to be position-invariant because there is no guarantee made that the way a vertex program might compute its homogenous position is exactly identical to the way conventional OpenGL transformation computes its homogenous positions. In a position-invariant vertex program, the homogeneous position (HPOS) is not output by the program. Instead, the OpenGL implementation is expected to compute the HPOS for position-invariant vertex programs in a manner exactly identical to how the homogenous position and window position are computed for a vertex by conventional OpenGL transformation. In this way position-invariant vertex programs guarantee correct multi-pass rendering semantics in cases where multiple passes are rendered and the second and subsequent passes use a GL_EQUAL depth test.

Issues

How should options to the vertex program semantics be handled?

RESOLUTION: A VP1.1 vertex program can contain a sequence of options. This extension provides a single option ("NV_position_invariant"). Specifying an option changes the way the program's subsequent instruction sequence are parsed, may add new semantic checks, and modifies the semantics by which the vertex program is executed.

Should this extension provide SUB and ABS instructions even though the functionality can be accomplished with ADD and MAX?

RESOLUTION: Yes. SUB and ABS provide no functionality that could not be accomplished in VP1.0 with ADD and MAX idioms, SUB and ABS provide more understanable vertex programs.

Should the optionalSign in a VP1.1 accept both "-" and "+"?

RESOLUTION: Yes. The "+" does not negate its operand but is available for symetry.

Is relative addressing available to position-invariant version 1.1 vertex programs?

RESOLUTION: No. This reflects a hardware restriction.

Should something be said about the relative performance of position-invariant vertex programs and conventional vertex programs?

RESOLUTION: For architectural reasons, position-invariant vertex programs may be _slightly_ faster than conventional vertex programs. This is true in the GeForce3 architecture. If your vertex program transforms the object-space position to clip-space with four DP4 instructions using the tracked GL_MODELVIEW_PROJECTION_NV matrix, consider using position-invariant vertex programs. Do not expect a measurable performance improvement unless vertex program processing is your bottleneck and your vertex program is relatively short.

Should position-invariant vertex programs have a lower limit on the maximum instructions?

RESOLUTION: Yes, the driver takes care to match the same instructions used for position transformation used by conventional transformation and this requires a few vertex program instructions.

New Procedures and Functions

None.

New Tokens

None.

Additions to Chapter 2 of the OpenGL 1.2.1 Specification (OpenGL Operation)

2.14.1.9 Vertex Program Register Accesses

Replace the first two sentences and update Table X.4:

"There are 21 vertex program instructions. The instructions and their respective input and output parameters are summarized in Table X.4."

Opcode	Inputs (scalar or vector)	Output (vector or replicated scalar)	Operation
ARL	s	address register	address register load
MOV	v	v	move
MUL	v,v	v	multiply
ADD	v,v	v	add
MAD	v,v,v	v	multiply and add
RCP	s	ssss	reciprocal
RSQ	s	ssss	reciprocal square root
DP3	v,v	ssss	3-component dot product
DP4	v,v	ssss	4-component dot product
DST	v,v	v	distance vector
MIN	v,v	v	minimum
MAX	v,v	v	maximum
SLT	v,v	v	set on less than
SGE	v,v	v	set on greater equal than
EXP	s	v	exponential base 2
LOG	s	v	logarithm base 2
LIT	v	v	light coefficients
DPH	v,v	ssss	homogeneous dot product
RCC	s	ssss	reciprocal clamped
SUB	v,v	v	subtract
ABS	v	v	absolute value

Table X.4: Summary of vertex program instructions. "v" indicates a vector input or output, "s" indicates a scalar input, and "ssss" indicates a scalar output replicated across a 4-component vector.

Add four new sections describing the DPH, RCC, SUB, and ABS instructions.

"2.14.1.10.18 DPH: Homogeneous Dot Product

The DPH instruction assigns the four-component dot product of the two source vectors where the W component of the first source vector is assumed to be 1.0 into the destination register.

```
t.x = source0.c***;
t.y = source0.*c**;
t.z = source0.**c*;
if (negate0) {
  t.x = -t.x;
  t.y = -t.y;
  t.z = -t.z;
}
u.x = source1.c***;
u.y = source1.*c**;
u.z = source1.**c*;
u.w = source1.***c;
if (negate1) {
  u.x = -u.x;
  u.y = -u.y;
  u.z = -u.z;
  u.w = -u.w;
}
v.x = t.x * u.x + t.y * u.y + t.z * u.z + u.w;
if (xmask) destination.x = v.x;
if (ymask) destination.y = v.x;
if (zmask) destination.z = v.x;
if (wmask) destination.w = v.x;
```

2.14.1.10.19 RCC: Reciprocal Clamped

The RCC instruction inverts the value of the source scalar, clamps the result as described below, and stores the clamped result into the destination register. The reciprocal of exactly 1.0 must be exactly 1.0.

Additionally (before clamping) the reciprocal of negative infinity gives [-0.0, -0.0, -0.0, -0.0]; the reciprocal of negative zero gives [-Inf, -Inf, -Inf, -Inf]; the reciprocal of positive zero gives [+Inf, +Inf, +Inf, +Inf]; and the reciprocal of positive infinity gives [0.0, 0.0, 0.0, 0.0].

```
t.x = source0.c;
if (negate0) {
  t.x = -t.x;
}
if (t.x == 1.0f) {
  u.x = 1.0f;
} else {
  u.x = 1.0f / t.x;
}
if (Positive(u.x)) {
  if (u.x > 1.884467e+019) {
    u.x = 1.884467e+019;   // the IEEE 32-bit binary value 0x5F800000
  } else if (u.x < 5.42101e-020) {
    u.x = 5.42101e-020;    // the IEEE 32-bit bindary value 0x1F800000
  }
} else {
  if (u.x < -1.884467e+019) {
    u.x = -1.884467e+019;  // the IEEE 32-bit binary value 0xDF800000
  } else if (u.x > -5.42101e-020) {
    u.x = -5.42101e-020;   // the IEEE 32-bit binary value 0x9F800000
  }
}
if (xmask) destination.x = u.x;
if (ymask) destination.y = u.x;
if (zmask) destination.z = u.x;
if (wmask) destination.w = u.x;
```

where Positive(x) is true for +0 and other positive values and false for -0 and other negative values; and

```
| u.x - IEEE(1.0f/t.x) | < 1.0f/(2^22)
```

for 1.0f <= t.x <= 2.0f. The intent of this precision requirement is that this amount of relative precision apply over all values of t.x."

2.14.1.10.20 SUB: Subtract

The SUB instruction subtracts the values of the one source vector from another source vector and stores the result into the destination register.

```
    t.x = source0.c***;
    t.y = source0.*c**;
    t.z = source0.**c*;
    t.w = source0.***c;
    if (negate0) {
      t.x = -t.x;
      t.y = -t.y;
      t.z = -t.z;
      t.w = -t.w;
    }
    u.x = source1.c***;
    u.y = source1.*c**;
    u.z = source1.**c*;
    u.w = source1.***c;
    if (negate1) {
      u.x = -u.x;
      u.y = -u.y;
      u.z = -u.z;
      u.w = -u.w;
    }
    if (xmask) destination.x = t.x - u.x;
    if (ymask) destination.y = t.y - u.y;
    if (zmask) destination.z = t.z - u.z;
    if (wmask) destination.w = t.w - u.w;
```

2.14.1.10.21 ABS: Absolute Value

The ABS instruction assigns the component-wise absolute value of a source vector into the destination register.

```
    t.x = source0.c***;
    t.y = source0.*c**;
    t.z = source0.**c*;
    t.w = source0.***c;
    if (xmask) destination.x = (t.x >= 0) ? t.x : -t.x;
    if (ymask) destination.y = (t.y >= 0) ? t.y : -t.y;
    if (zmask) destination.z = (t.z >= 0) ? t.z : -t.z;
    if (wmask) destination.w = (t.w >= 0) ? t.w : -t.w;
```

Insert sections 2.14.A and 2.14.B after section 2.14.4

"2.14.A Version 1.1 Vertex Programs

Version 1.1 vertex programs provide support for the DPH, RCC, SUB, and ABS instructions (see sections 2.14.1.10.18 through 2.14.1.10.21).

Version 1.1 vertex programs are loaded with the LoadProgramNV command (see section 2.14.1.7). The target must be VERTEX_PROGRAM_NV to load a version 1.1 vertex program. The initial "!!VP1.1" token designates the program should be parsed and treated as a version 1.1 vertex program.

Version 1.1 programs must conform to a more expanded grammar than the grammar for vertex programs. The version 1.1 vertex program grammar for syntactically valid sequences is the same as the grammar defined in section 2.14.1.7 with the following modified rules:

```
<program>              ::= "!!VP1.1" <optionSequence> <instructionSequence> "END"

<optionSequence>       ::= <optionSequence> <option>
                         | ""

<option>               ::= "OPTION" "NV_position_invariant" ";"

<VECTORop>             ::= "MOV"
                         | "LIT"
                         | "ABS"

<SCALARop>             ::= "RCP"
                         | "RSQ"
                         | "EXP"
                         | "LOG"
                         | "RCC"

<BINop>                ::= "MUL"
                         | "ADD"
                         | "DP3"
                         | "DP4"
                         | "DST"
                         | "MIN"
                         | "MAX"
                         | "SLT"
                         | "SGE"
                         | "DPH"
                         | "SUB"

<optionalSign>         ::= "-"
                         | "+"
                         | ""
```

Except for supporting the additional DPH, RCC, SUB, and ABS instructions, version 1.1 vertex programs with no options specified otherwise behave in the same manner as version 1.0 vertex programs.

2.14.B Position-invariant Vertex Program Option

By default, vertex programs are _not_ guaranteed to be position-invariant because there is no guarantee made that the way a vertex program might compute its homogenous position is exactly identical to the way conventional OpenGL transformation computes its homogenous positions. However in a position-invariant vertex program, the homogeneous position (HPOS) is not output by the program. Instead, the OpenGL implementation is expected to compute the HPOS for position-invariant vertex programs in a manner exactly identical to how the homogenous position and window position are computed for a vertex by conventional OpenGL transformation (assuming vertex weighting and vertex blending are disabled). In this way position-invariant vertex programs guarantee correct multi-pass rendering semantics in cases where multiple passes are rendered with conventional OpenGL transformation and position-invariant vertex programs and the second and subsequent passes use a EQUAL depth test.

    If an <option> with the identifier "NV_position_invariant" is
    encountered during the parsing of the program, the specified program
    is presumed to be position-invariant.

    When a position-invariant vertex program is specified, the
    <vertexResultRegName> rule is replaced with the following rule
    (that does not provide "HPOS"):

```
    <vertexResultRegName>  ::= "COL0"
                             | "COL1"
                             | "BFC0"
                             | "BFC1"
                             | "FOGC"
                             | "PSIZ"
                             | "TEX0"
                             | "TEX1"
                             | "TEX2"
                             | "TEX3"
                             | "TEX4"
                             | "TEX5"
                             | "TEX6"
                             | "TEX7"
```

    While position-invariant version 1.1 vertex programs provide
    position-invariance, such programs do not provide support for
    relative program parameter addressing. The <relProgParamReg> rule
    for version 1.1 position-invariant vertex programs is replaced by
    (eliminating the relative addressing cases):

```
    <relProgParamReg>      ::= "c" "[" <addrReg> "]"
```

    Note that while the ARL instruction is still available to
    position-invariant version 1.1 vertex programs, it provides no
    meaningful functionality without support for relative addressing.

    The semantic restriction for vertex program instruction length is
    changed in the case of position-invariant vertex programs to the
    following: A position-invariant vertex program fails to load if it
    contains more than 124 instructions.

    "

Additions to Chapter 4 of the OpenGL 1.2.1 Specification (Per-Fragment Operations and the Framebuffer)

    None

Additions to Chapter 5 of the OpenGL 1.2.1 Specification (Special Functions)

    None

Additions to Chapter 6 of the OpenGL 1.2.1 Specification (State and State Requests)

    None

Additions to the AGL/GLX/WGL Specifications

None

GLX Protocol

None

Errors

None

New State

None

Name

    NV_vertex_program2

Name Strings

    GL_NV_vertex_program2

Notice

IP Status

    NVIDIA Proprietary.

Status

    Implemented in CineFX (NV30) Emulation driver, August 2002.
    Shipping in Release 40 NVIDIA driver for CineFX hardware, January 2003.

Version

    Last Modified Date: $Date: 2003/05/12 $
    NVIDIA Revision: Revision: #30

Number

    287

Dependencies

Written based on the wording of the OpenGL 1.3 Specification and requires OpenGL 1.3.

Written based on the wording of the NV_vertex_program extension specification, version 1.0.

NV_vertex_program is required.

Overview

This extension further enhances the concept of vertex programmability introduced by the NV_vertex_program extension, and extended by NV_vertex_program1_1. These extensions create a separate vertex program mode where the configurable vertex transformation operations in unextended OpenGL are replaced by a user-defined program.

This extension introduces the VP2 execution environment, which extends the VP1 execution environment introduced in NV_vertex_program. The VP2 environment provides several language features not present in previous vertex programming execution environments:

* Branch instructions allow a program to jump to another instruction specified in the program.

  * Branching support allows for up to four levels of subroutine calls/returns.

  * A four-component condition code register allows an application to compute a component-wise write mask at run time and apply that mask to register writes.

  * Conditional branches are supported, where the condition code register is used to determine if a branch should be taken.

  * Programmable user clipping is supported support (via the CLP0-CLP5 clip distance registers). Primitives are clipped to the area where the interpolated clip distances are greater than or equal to zero.

  * Instructions can perform a component-wise absolute value operation on any operand load.

The VP2 execution environment provides a number of new instructions, and extends the semantics of several instructions already defined in NV_vertex_program.

  * ARR: Operates like ARL, except that float-to-int conversion is done by rounding. Equivalent results could be achieved (less efficiently) in NV_vertex program using an ADD/ARL sequence and a program parameter holding the value 0.5.

  * BRA, CAL, RET: Branch, subroutine call, and subroutine return instructions.

  * COS, SIN: Adds support for high-precision sine and cosine computations.

  * FLR, FRC: Adds support for computing the floor and fractional portion of floating-point vector components. Equivalent results could be achieved (less efficiently) in NV_vertex_program using the EXP instruction to compute the fractional portion of one component at a time.

  * EX2, LG2: Adds support for high-precision exponentiation and logarithm computations.

  * ARA: Adds pairs of components of an address register; useful for looping and other operations.

  * SEQ, SFL, SGT, SLE, SNE, STR: Add six new "set on" instructions, similar to the SLT and SGE instructions defined in NV_vertex_program. Equivalent results could be achieved (less efficiently) in NV_vertex_program with multiple SLT, SGE, and arithmetic instructions.

  * SSG: Adds a new "set sign" operation, which produces a vector holding negative one for negative components, zero for components with a value of zero, and positive one for positive components. Equivalent results could be achieved (less efficiently) in NV_vertex_program with multiple SLT, SGE, and arithmetic instructions.

  * The ARL instruction is extended to operate on four components instead of a single component.

* All instructions that produce integer or floating-point result vectors have variants that update the condition code register based on the result vector.

This extension also raises some of the resource limitations in the NV_vertex_program extension.

* 256 program parameter registers (versus 96 in NV_vertex_program).

* 16 temporary registers (versus 12 in NV_vertex_program).

* Two four-component integer address registers (versus one single-component register in NV_vertex_program).

* 256 total vertex program instructions (versus 128 in NV_vertex_program).

* Including loops, programs can execute up to 64K instructions.

Issues

This extension builds upon the NV_vertex_program extension. Should this specification contain selected edits to the NV_vertex_program specification or should the specs be unified?

RESOLVED: Since NV_vertex_program and NV_vertex_program2 programs share many features, the main section of this specification is unified and describes both types of programs. Other sections containing NV_vertex_program features that are unchanged by this extension will not be edited.

How can a program use condition codes to avoid extra computations?

Consider the example of evaluating the OpenGL lighting model for a given light. If the diffuse dot product is negative (roughly 1/2 the time for random geometry), the only contribution to the light is ambient. In this case, condition codes and branching can skip over a number of unneeded instructions.

```
        # R0 holds accumulated light color
        # R2 holds normal
        # R3 holds computed light vector
        # R4 holds computed half vector
        # c[0] holds ambient light/material product
        # c[1] holds diffuse light/material product
        # c[2].xyz holds specular light/material product
        # c[2].w   holds specular exponent
        DP3C R1.x, R2, R3;            # diffuse dot product
        ADD  R0, R0, c[0];            # accumulate ambient
        BRA  pointsAway (LT.x)        # skip rest if diffuse dot < 0
        MOV  R1.w, c[2].w;
        DP3  R1.y, R2, R4;            # specular dot product
        LIT  R1, R1;                  # compute expontiated specular
        MAD  R4, c[1], R0.y;          # accumulate diffuse
        MAD  R4, c[2], R0.z;          # accumulate specular
      pointsAway:
        ...                           # continue execution
```

How can a program use subroutines and branch tables?

With subroutines, a program can encapsulate a small piece of functionality into a subroutine and call it multiple times, as in CPU code. Applications will need to identify the registers used to pass data to and from the subroutine.

Subroutines could be used for applications like evaluating lighting equations for a single light. With conditional branching and subroutines, a variable number of lights (which could even vary per-vertex) can be easily supported.

```
      accumulate:
        # R0 holds the accumulated result
        # R1 holds the value to add
        ADD R0, R1;
        RET;

        # Compute floor(A)*B by repeated addition using a subroutine.  Yes,
        # this is a stupid example.
        #
        # c[0] holds (A,B,0,1).
        # R0 holds the accumulated result
        # R1 holds B, the value to accumulate.
        # R2 holds the number of iterations remaining.
        MOV R0, c[0].z;               # start with zero
        MOV R1, c[0].y;
        FLRC R2.x, c[0].x;
        BRA done (LE.x);
      top:
        CAL accumulate;
        ADDC R2.x, R2.x, -c[0].w;     # decrement count
        BRA top (GT.x);
      done:
        ...
```

How can conventional OpenGL clip planes be supported in vertex programs?

The clip distance in the OpenGL specification can be evaluated with a simple DP4 instruction that writes to one of the six clip distance registers. Primitives will automatically be clipped to the half-space where o[CLPx] >= 0, which matches the definition in the spec.

```
    # R0 holds eye coordinates
    # c[0] holds eye-space clip plane coefficients
    DP4 o[CLP0].x, R0, c[0];
```

Note that the clip plane or clip distance volume corresponding to the o[CLPn] register used must be enabled, or no clipping will be performed.

The clip distance registers allow for clip distance volumes to be computed more-or-less arbitrarily. To approximate clipping to a sphere of radius <n>, the following code can be used.

```
    # R0 holds eye coordinates
    # c[0].xyz holds sphere center
    # c[0].w holds the square of the sphere radius
    SUB R1.xyz, R0, c[0];           # distance vector
    DP3 R1.w, R1, R1;               # compute distance squared
    SUB o[CLP0].x, c[0].w, R1.w;    # compute r^2 - d^2
```

Since the clip distance is interpolated linearly over a primitive, the clip distance evaluated at a point will represent a piecewise-linear approximation of the true distance. The approximation will become increasingly more accurate as the primitive is tesselated more finely.

How can looping be achieved in vertex programs?

Simple loops can be achieved using a general purpose floating-point register component as a counter. The following code calls a function named "function" <n> times, where <n> is specified in a program parameter register component.

```
    # c[0].x holds the number of iterations to execute.
    # c[1].x holds the constant 1.0.
    MOVC R15.x, c[0].x;
  startLoop:
    CAL  function (GT.x);            # if (counter > 0) function();
    SUBC R15.x, R15.x, c[1].x;       # counter = counter - 1;
    BRA  startLoop (GT.x);           # if (counter > 0) goto start;
  endLoop:
    ...
```

More complex loops (where a separate index may be needed for indexed addressing into the program parameter array) can be achieved using the ARA instruction, which will add the x/z and y/w components of an address register.

```
        # c[0].x holds the number of iterations to execute
        # c[0].y holds the initial index value
        # c[0].z holds the constant -1.0 (used for the iteration count)
        # c[0].w holds the index step value
        ARLC A1, c[0];
      startLoop:
        CAL  function (GT.x);            # if (counter > 0) function();
                                         # Note: A1.y can be used for
                                         # indexing in function().
        ARAC A1.xy, A1;                  # counter = counter - 1;
                                         # index += loopStep;
        BRA  startLoop (GT.x);           # if (counter > 0) goto start;
      endLoop:
        ...
```

Should this specification add support for vertex state programs beyond the VP1 execution environment?

No. Vertex state programs are a little-used feature of NV_vertex_program and don't perform particularly well. They are still supported for compatibility with the original NV_vertex_program spec, but they will not be extended to support new features.

How are NaN's be handled in the "set on" instructions (SEQ, SGE, SGT, SLE, SLT, SNE)? What about MIN, MAX? SSG? When doing condition code tests?

Any of these instructions involving a NaN operand will produce a NaN result. This behavior differs from the NV_fragment_program extension. There, SEQ, SGE, SGT, SLE, and SLT will produce 0.0 if either operand is a NaN, and SNE will produce 1.0 if either operand is a NaN.

For condition code updates, NaN values will result in "UN" condition codes. All conditionals using a "UN" condition code, except "TR" and "NE" will evaluate to false. This behavior is identical to the functionality in NV_fragment_program.

How can the various features of this extension be used to provide skinning functionality similar to that in ARB_vertex_blend and ARB_matrix_palette? And how can that functionality be extended?

Assume an implementation that allows application of up to 8 matrices at once. Further assume that v[12].xyzw and v[13].xyzw hold the set of 8 weights, and v[14].xyzw and v[15].xyzw hold the set of 8 matrix indices. Furthermore, assume that the palette of matrices are stored/tracked at c[0], c[4], c[8], and so on. As an additional optimization, an application can specify that fewer than 8 matrices should be applied by storing a negative palette index immediately after the last index is applied.

Skinning support in this example can be provided by the following code:

```
          ARLC A0, v[14];                 # load 4 palette indices at once
          DP4 R1.x, c[A0.x+0], v[0];      # 1st matrix transform
          DP4 R1.y, c[A0.x+1], v[0];
          DP4 R1.z, c[A0.x+2], v[0];
          DP4 R1.w, c[A0.x+3], v[0];
          MUL R0, R1, v[12].x;            # accumulate weighted sum in R0
          BRA end (LT.y);                 # stop on a negative matrix index
          DP4 R1.x, c[A0.y+0], v[0];      # 2nd matrix transform
          DP4 R1.y, c[A0.y+1], v[0];
          DP4 R1.z, c[A0.y+2], v[0];
          DP4 R1.w, c[A0.y+3], v[0];
          MAD R0, R1, v[12].y, R0;        # accumulate weighted sum in R0
          BRA end (LT.z);                 # stop on a negative matrix index

          ...                             # 3rd and 4th matrix transform

          ARLC A0, v[15];                 # load next four palette indices
          BRA end (LT.x);
          DP4 R1.x, c[A0.x+0], v[0];      # 5th matrix transform
          DP4 R1.y, c[A0.x+1], v[0];
          DP4 R1.z, c[A0.x+2], v[0];
          DP4 R1.w, c[A0.x+3], v[0];
          MAD R0, R1, v[13].x, R0;        # accumulate weighted sum in R0
          BRA end (LT.y);                 # stop on a negative matrix index

          ...                             # 6th, 7th, and 8th matrix transform

        end:
          ...                             # any additional instructions
```

      The amount of code used by this example could further be reduced using a
      subroutine performing four transformations at a time:

```
          ARLC A0, v[14];  # load first four indices
          CAL  skin4;      # do first four transformations
          BRA  end (LT);   # end if any of the first 4 indices was < 0
          ARLC A0, v[15];  # load second four indices
          CAL  skin4;      # do second four transformations
        end:
          ...              # any additional instructions
```

Why does the RCC instruction exist?

RESOLVED: To perform numeric operations that will avoid overflow and underflow issues.

Should the specification provide more examples?

RESOLVED: It would be nice.

New Procedures and Functions

None.

New Tokens

None.

Additions to Chapter 2 of the OpenGL 1.3 Specification (OpenGL Operation)

Modify Section 2.11, Clipping (p. 39)

(modify last paragraph, p. 39) When the GL is not in vertex program mode

(section 2.14), this view volume may be further restricted by as many as n client-defined clip planes to generate the clip volume. ...

(add before next-to-last paragraph, p. 40) When the GL is in vertex program mode, the view volume may be restricted to the individual clip distance volumes derived from the per-vertex clip distances (o[CLP0] - o[CLP5]). Clip distance volumes are applied if and only if per-vertex clip distances are not supported in the vertex program execution environment. A point P belonging to the primitive under consideration is in the clip distance volume numbered n if and only if

```
  c_n(P) >= 0,
```

where c_n(P) is the interpolated value of the clip distance CLPn at the point P. For point primitives, c_n(P) is simply the clip distance for the vertex in question. For line and triangle primitives, per-vertex clip distances are interpolated using a weighted mean, with weights derived according to the algorithms described in sections 3.4 and 3.5.

(modify next-to-last paragraph, p.40) Client-defined clip planes or clip distance volumes are enabled with the generic Enable command and disabled with the Disable command. The value of the argument to either command is CLIP PLANEi where i is an integer between 0 and n; specifying a value of i enables or disables the plane equation with index i. The constants obey CLIP PLANEi = CLIP PLANE0 + i.

Add Section 2.14, Vertex Programs (p. 57). This section supersedes the similar section added in the NV_vertex_program extension and extended in the NV_vertex_program1_1 extension.

The conventional GL vertex transformation model described in sections 2.10 through 2.13 is a configurable, but essentially hard-wired, sequence of per-vertex computations based on a canonical set of per-vertex parameters and vertex transformation related state such as transformation matrices, lighting parameters, and texture coordinate generation parameters.

The general success and utility of the conventional GL vertex transformation model reflects its basic correspondence to the typical vertex transformation requirements of 3D applications.

However when the conventional GL vertex transformation model is not sufficient, the vertex program mode provides a substantially more flexible model for vertex transformation. The vertex program mode permits applications to define their own vertex programs.

Section 2.14.1, Vertex Program Execution Environment

The vertex program execution environment is an operational model that defines how a program is executed. The execution environment includes a set of instructions, a set of registers, and semantic rules defining how operations are performed. There are three vertex program execution environments, VP1, VP1.1, and VP2. The environment names are taken from the mandatory program prefix strings found at the beginning of all vertex programs. The VP1.1 execution environment is a minor addition to the VP1 execution environment, so references to the VP1 execution environment below apply to both VP1 and VP1.1 execution environments except where otherwise noted.

The vertex program instruction set consists primarily of floating-point 4-component vector operations operating on per-vertex attributes and program parameters. Vertex programs execute on a per-vertex basis and operate on each vertex completely independently from the processing of other vertices. Vertex programs execute without data hazards so results computed in one operation can be used immediately afterwards. Vertex programs produce a set of vertex result vectors that becomes the set of transformed vertex parameters used by primitive assembly.

In the VP1 environment, vertex programs execute a finite fixed sequence of instructions with no branching or looping. In the VP2 environment, vertex programs support conditional and unconditional branches and four levels of subroutine calls.

The vertex program register set consists of six types of registers described in the following sections.

Section 2.14.1.1, Vertex Attribute Registers

The Vertex Attribute Registers are sixteen 4-component vector floating-point registers containing the current vertex's per-vertex attributes. These registers are numbered 0 through 15. These registers are private to each vertex program invocation and are initialized at each vertex program invocation by the current vertex attribute state specified with VertexAttribNV commands. These registers are read-only during vertex program execution. The VertexAttribNV commands used to update the vertex attribute registers can be issued both outside and inside of Begin/End pairs. Vertex program execution is provoked by updating vertex attribute zero. Updating vertex attribute zero outside of a Begin/End pair is ignored without generating any error (identical to the Vertex command operation).

The commands

```
  void VertexAttrib{1234}{sfd}NV(uint index, T coords);
  void VertexAttrib{1234}{sfd}vNV(uint index, T coords);
  void VertexAttrib4ubNV(uint index, T coords);
  void VertexAttrib4ubvNV(uint index, T coords);
```

specify the particular current vertex attribute indicated by index. The coordinates for each vertex attribute are named x, y, z, and w. The VertexAttrib1NV family of commands sets the x coordinate to the provided single argument while setting y and z to 0 and w to 1. Similarly, VertexAttrib2NV sets x and y to the specified values, z to 0 and w to 1; VertexAttrib3NV sets x, y, and z, with w set to 1, and VertexAttrib4NV sets all four coordinates. The error INVALID_VALUE is generated if index is greater than 15.

No conversions are applied to the vertex attributes specified as type short, float, or double. However, vertex attributes specified as type ubyte are converted as described by Table 2.6.

The commands

```
  void VertexAttribs{1234}{sfd}vNV(uint index, sizei n, T coords[]);
  void VertexAttribs4ubvNV(uint index, sizei n, GLubyte coords[]);
```

specify a contiguous set of n vertex attributes. The effect of

```
  VertexAttribs{1234}{sfd}vNV(index, n, coords)
```

is the same (assuming no errors) as the command sequence

```
  #define NUM k  /* where k is 1, 2, 3, or 4 components */
  int i;
  for (i=n-1; i>=0; i--) {
    VertexAttrib{NUM}{sfd}vNV(i+index, &coords[i*NUM]);
  }
```

VertexAttribs4ubvNV behaves similarly.

The VertexAttribNV calls equivalent to VertexAttribsNV are issued in reverse order so that vertex program execution is provoked when index is zero only after all the other vertex attributes have first been specified.

The set and operation of vertex attribute registers are identical for both VP1 and VP2 execution environment.

Section 2.14.1.2, Program Parameter Registers

The Program Parameter Registers are a set of 4-component floating-point vector registers containing the vertex program parameters. In the VP1 execution environment, there are 96 registers, numbered 0 through 95. In the VP2 execution environment, there are 256 registers, numbered 0 through 255. This relatively large set of registers is intended to hold parameters such as matrices, lighting parameters, and constants required by vertex programs. Vertex program parameter registers can be updated in one of two ways: by the ProgramParameterNV commands outside of a Begin/End pair or by a vertex state program executed outside of a Begin/End pair (vertex state programs are discussed in section 2.14.3).

The commands

```
  void ProgramParameter4fNV(enum target, uint index,
                            float x, float y, float z, float w)
  void ProgramParameter4dNV(enum target, uint index,
                            double x, double y, double z, double w)
```

specify the particular program parameter indicated by index. The coordinates values x, y, z, and w are assigned to the respective components of the particular program parameter. target must be VERTEX_PROGRAM_NV.

The commands

```
  void ProgramParameter4dvNV(enum target, uint index, double *params);
  void ProgramParameter4fvNV(enum target, uint index, float *params);
```

operate identically to ProgramParameter4fNV and ProgramParameter4dNV respectively except that the program parameters are passed as an array of four components.

The error INVALID_VALUE is generated if the specified index is greater than or equal to the number of program parameters in the execution environment (96 for VP1, 256 for VP2).

The commands

```
  void ProgramParameters4dvNV(enum target, uint index,
                              uint num, double *params);
  void ProgramParameters4fvNV(enum target, uint index,
                              uint num, float *params);
```

specify a contiguous set of num program parameters. The effect is the same (assuming no errors) as

```
  for (i=index; i<index+num; i++) {
    ProgramParameter4{fd}vNV(target, i, &params[i*4]);
  }
```

The error INVALID_VALUE is generated if sum of <index> and <num> is greater than the number of program parameters in the execution environment (96 for VP1, 256 for VP2).

The program parameter registers are shared to all vertex program invocations within a rendering context. ProgramParameterNV command updates and vertex state program executions are serialized with respect to vertex program invocations and other vertex state program executions.

Writes to the program parameter registers during vertex state program execution can be maskable on a per-component basis.

The initial value of all 96 (VP1) or 256 (VP2) program parameter registers is (0,0,0,0).

Section 2.14.1.3, Address Registers

The Address Registers are 4-component vector registers with signed 10-bit integer components. In the VP1 execution environment, there is only a single address register (A0) and only the x component of the register is accessible. In the VP2 execution environment, there are two address registers (A0 and A1), of which all four components are accessible. The address registers are private to each vertex program invocation and are initialized to (0,0,0,0) at every vertex program invocation. These registers can be written during vertex program execution (but not read) and their values can be used for as a relative offset for reading vertex program parameter registers. Only the vertex program parameter registers can be read using relative addressing (writes using relative addressing are not supported).

See the discussion of relative addressing of program parameters in section 2.14.2.1 and the discussion of the ARL instruction in section 2.14.3.4.

Section 2.14.1.4, Temporary Registers

The Temporary Registers are 4-component floating-point vector registers used to hold temporary results during vertex program execution. In the VP1 execution environment, there are 12 temporary registers, numbered 0 through 11. In the VP2 execution environment, there are 16 temporary registers, numbered 0 through 15. These registers are private to each vertex program invocation and initialized to (0,0,0,0) at every vertex program invocation. These registers can be read and written during vertex program execution. Writes to these registers can be maskable on a per-component basis.

In the VP2 execution environment, there is one additional temporary pseudo-register, "CC". CC is treated as unnumbered, write-only temporary register, whose sole purpose is to allow instructions to modify the condition code register (section 2.14.1.6) without overwriting the contents of any temporary register.

Section 2.14.1.5, Vertex Result Registers

The Vertex Result Registers are 4-component floating-point vector registers used to write the results of a vertex program. There are 15 result registers in the VP1 execution environment, and 21 in the VP2 execution environment. Each register value is initialized to (0,0,0,1) at the invocation of each vertex program. Writes to the vertex result registers can be maskable on a per-component basis. These registers are named in Table X.1 and further discussed below.

Vertex Result Register Name	Description	Component Interpretation
HPOS	Homogeneous clip space position	(x,y,z,w)
COL0	Primary color (front-facing)	(r,g,b,a)
COL1	Secondary color (front-facing)	(r,g,b,a)
BFC0	Back-facing primary color	(r,g,b,a)
BFC1	Back-facing secondary color	(r,g,b,a)
FOGC	Fog coordinate	(f,*,*,*)
PSIZ	Point size	(p,*,*,*)
TEX0	Texture coordinate set 0	(s,t,r,q)
TEX1	Texture coordinate set 1	(s,t,r,q)
TEX2	Texture coordinate set 2	(s,t,r,q)
TEX3	Texture coordinate set 3	(s,t,r,q)
TEX4	Texture coordinate set 4	(s,t,r,q)
TEX5	Texture coordinate set 5	(s,t,r,q)
TEX6	Texture coordinate set 6	(s,t,r,q)
TEX7	Texture coordinate set 7	(s,t,r,q)
CLP0(*)	Clip distance 0	(d,*,*,*)
CLP1(*)	Clip distance 1	(d,*,*,*)
CLP2(*)	Clip distance 2	(d,*,*,*)
CLP3(*)	Clip distance 3	(d,*,*,*)
CLP4(*)	Clip distance 4	(d,*,*,*)
CLP5(*)	Clip distance 5	(d,*,*,*)

Table X.1: Vertex Result Registers. (*) Registers CLP0 through CLP5, are available only in the VP2 execution environment.

HPOS is the transformed vertex's homogeneous clip space position. The vertex's homogeneous clip space position is converted to normalized device coordinates and transformed to window coordinates as described at the end of section 2.10 and in section 2.11. Further processing (subsequent to vertex program termination) is responsible for clipping primitives assembled from vertex program-generated vertices as described in section 2.10 but all client-defined clip planes are treated as if they are disabled when vertex program mode is enabled.

Four distinct color results can be generated for each vertex. COL0 is the transformed vertex's front-facing primary color. COL1 is the transformed vertex's front-facing secondary color. BFC0 is the transformed vertex's back-facing primary color. BFC1 is the transformed vertex's back-facing secondary color.

Primitive coloring may operate in two-sided color mode. This behavior is enabled and disabled by calling Enable or Disable with the symbolic value VERTEX_PROGRAM_TWO_SIDE_NV. The selection between the back-facing colors and the front-facing colors depends on the primitive of which the vertex is a part. If the primitive is a point or a line segment, the front-facing colors are always selected. If the primitive is a polygon and two-sided color mode is disabled, the front-facing colors are selected. If it is a polygon and two-sided color mode is enabled, then the selection is based on the sign of the (clipped or unclipped) polygon's signed area computed in window coordinates. This facingness determination is identical to the two-sided lighting facingness determination described in section 2.13.1.

The selected primary and secondary colors for each primitive are clamped to the range [0,1] and then interpolated across the assembled primitive during rasterization with at least 8-bit accuracy for each color component.

FOGC is the transformed vertex's fog coordinate. The register's first floating-point component is interpolated across the assembled primitive during rasterization and used as the fog distance to compute per-fragment the fog factor when fog is enabled. However, if both fog and vertex program mode are enabled, but the FOGC vertex result register is not written, the fog factor is overridden to 1.0. The register's other three components are ignored.

Point size determination may operate in program-specified point size mode. This behavior is enabled and disabled by calling Enable or Disable with the symbolic value VERTEX_PROGRAM_POINT_SIZE_NV. If the vertex is for a point primitive and the mode is enabled and the PSIZ vertex result is written, the point primitive's size is determined by the clamped x component of the PSIZ register. Otherwise (because vertex program mode is disabled, program-specified point size mode is disabled, or because the vertex program did not write PSIZ), the point primitive's size is determined by the point size state (the state specified using the PointSize command).

The PSIZ register's x component is clamped to the range zero through either the hi value of ALIASED_POINT_SIZE_RANGE if point smoothing is disabled or the hi value of the SMOOTH_POINT_SIZE_RANGE if point smoothing is enabled. The register's other three components are ignored.

If the vertex is not for a point primitive, the value of the PSIZ vertex result register is ignored.

TEX0 through TEX7 are the transformed vertex's texture coordinate sets for texture units 0 through 7. These floating-point coordinates are interpolated across the assembled primitive during rasterization and used for accessing textures. If the number of texture units supported is less than eight, the values of vertex result registers that do not correspond to existent texture units are ignored.

CLP0 through CLP5, available only in the VP2 execution environment, are the transformed vertex's clip distances. These floating-point coordinates are used by post-vertex program clipping process (see section 2.11).

Section 2.14.1.6, The Condition Code Register

The VP2 execution environment provides a single four-component vector called the condition code register. Each component of this register is one of four enumerated values: GT (greater than), EQ (equal), LT (less than), or UN (unordered). The condition code register can be used to mask writes to registers and to evaluate conditional branches.

Most vertex program instructions can optionally update the condition code register. When a vertex program instruction updates the condition code register, a condition code component is set to LT if the corresponding component of the result is less than zero, EQ if it is equal to zero, GT if it is greater than zero, and UN if it is NaN (not a number).

The condition code register is initialized to a vector of EQ values each time a vertex program executes.

There is no condition code register available in the VP1 execution environment.

Section 2.14.1.7, Semantic Meaning for Vertex Attributes and Program Parameters

One important distinction between the conventional GL vertex transformation mode and the vertex program mode is that per-vertex parameters and other state parameters in vertex program mode do not have dedicated semantic interpretations the way that they do with the conventional GL vertex transformation mode.

For example, in the conventional GL vertex transformation mode, the Normal command specifies a per-vertex normal. The semantic that the Normal command supplies a normal for lighting is established because that is how the per-vertex attribute supplied by the Normal command is used by the conventional GL vertex transformation mode. Similarly, other state parameters such as a light source position have semantic interpretations based on how the conventional GL vertex transformation model uses each particular parameter.

In contrast, vertex attributes and program parameters for vertex programs have no pre-defined semantic meanings. The meaning of a vertex attribute or program parameter in vertex program mode is defined by how the vertex attribute or program parameter is used by the current vertex program to compute and write values to the Vertex Result Registers. This is the reason that per-vertex attributes and program parameters for vertex programs are numbered instead of named.

For convenience however, the existing per-vertex parameters for the conventional GL vertex transformation mode (vertices, normals, colors, fog coordinates, vertex weights, and texture coordinates) are aliased to numbered vertex attributes. This aliasing is specified in Table X.2. The table includes how the various conventional components map to the 4-component vertex attribute components.

Vertex Attribute Register Number	Conventional Per-vertex Parameter	Conventional Per-vertex Parameter Command	Conventional Component Mapping
0	vertex position	Vertex	x,y,z,w
1	vertex weights	VertexWeightEXT	w,0,0,1
2	normal	Normal	x,y,z,1
3	primary color	Color	r,g,b,a
4	secondary color	SecondaryColorEXT	r,g,b,1
5	fog coordinate	FogCoordEXT	fc,0,0,1
6	-	-	-
7	-	-	-
8	texture coord 0	MultiTexCoord(GL_TEXTURE0_ARB, ...)	s,t,r,q
9	texture coord 1	MultiTexCoord(GL_TEXTURE1_ARB, ...)	s,t,r,q
10	texture coord 2	MultiTexCoord(GL_TEXTURE2_ARB, ...)	s,t,r,q
11	texture coord 3	MultiTexCoord(GL_TEXTURE3_ARB, ...)	s,t,r,q
12	texture coord 4	MultiTexCoord(GL_TEXTURE4_ARB, ...)	s,t,r,q
13	texture coord 5	MultiTexCoord(GL_TEXTURE5_ARB, ...)	s,t,r,q
14	texture coord 6	MultiTexCoord(GL_TEXTURE6_ARB, ...)	s,t,r,q
15	texture coord 7	MultiTexCoord(GL_TEXTURE7_ARB, ...)	s,t,r,q

Table X.2: Aliasing of vertex attributes with conventional per-vertex parameters.

Only vertex attribute zero is treated specially because it is the attribute that provokes the execution of the vertex program; this is the attribute that aliases to the Vertex command's vertex coordinates.

The result of a vertex program is the set of post-transformation vertex parameters written to the Vertex Result Registers. All vertex programs must write a homogeneous clip space position, but the other Vertex Result Registers can be optionally written.

Clipping and culling are not the responsibility of vertex programs because these operations assume the assembly of multiple vertices into a primitive. View frustum clipping is performed subsequent to vertex program execution. Clip planes are not supported in the VP1 execution environment. Clip planes are supported indirectly via the clip distance (o[CLPx]) registers in the VP2 execution environment.

Section 2.14.1.8, Vertex Program Specification

Vertex programs are specified as an array of ubytes. The array is a string of ASCII characters encoding the program.

The command

```
  LoadProgramNV(enum target, uint id, sizei len,
                const ubyte *program);
```

loads a vertex program when the target parameter is VERTEX_PROGRAM_NV. Multiple programs can be loaded with different names. id names the program to load. The name space for programs is the positive integers (zero is reserved). The error INVALID_VALUE occurs if a program is loaded with an id of zero. The error INVALID_OPERATION is generated if a program is loaded for an id that is currently loaded with a program of a different program target. Managing the program name space and binding to vertex programs is discussed later in section 2.14.1.8.

program is a pointer to an array of ubytes that represents the program being loaded. The length of the array is indicated by len.

A second program target type known as vertex state programs is discussed in 2.14.4.

At program load time, the program is parsed into a set of tokens possibly separated by white space. Spaces, tabs, newlines, carriage returns, and comments are considered whitespace. Comments begin with the character "#" and are terminated by a newline, a carriage return, or the end of the program array.

The Backus-Naur Form (BNF) grammar below specifies the syntactically valid sequences for several types of vertex programs. The set of valid tokens can be inferred from the grammar. The token "" represents an empty string and is used to indicate optional rules. A program is invalid if it contains any undefined tokens or characters.

The grammar provides for three different vertex program types, corresponding to the three vertex program execution environments. VP1, VP1.1, and VP2 programs match the grammar rules <vp1-program>, <vp11-program>, and <vp2-program>, respectively. Some grammar rules correspond to features or instruction forms available only in certain execution environments. Rules beginning with the prefix "vp1-" are available only to VP1 and VP1.1 programs. Rules beginning with the prefixes "vp11-" and "vp2-" are available only to VP1.1 and VP2 programs, respectively.

```
<program>              ::= <vp1-program>
                         | <vp11-program>
                         | <vp2-program>

<vp1-program>          ::= "!!VP1.0" <programBody> "END"

<vp11-program>         ::= "!!VP1.1" <programBody> "END"

<vp2-program>          ::= "!!VP2.0" <programBody> "END"

<programBody>          ::= <optionSequence> <programText>

<optionSequence>       ::= <option> <optionSequence>
                         | ""

<option>               ::= "OPTION" <vp11-option> ";"
                         | "OPTION" <vp2-option> ";"

<vp11-option>          ::= "NV_position_invariant"

<vp2-option>           ::= "NV_position_invariant"

<programText>          ::= <programTextItem> <programText>
                         | ""
```

```
    <programTextItem>      ::= <instruction> ";"
                             | <vp2-instructionLabel>

    <instruction>          ::= <ARL-instruction>
                             | <VECTORop-instruction>
                             | <SCALARop-instruction>
                             | <BINop-instruction>
                             | <TRIop-instruction>
                             | <vp2-BRA-instruction>
                             | <vp2-RET-instruction>
                             | <vp2-ARA-instruction>

    <ARL-instruction>      ::= <vp1-ARL-instruction>
                             | <vp2-ARL-instruction>

    <vp1-ARL-instruction>  ::= "ARL" <maskedAddrReg> "," <scalarSrc>

    <vp2-ARL-instruction>  ::= <vp2-ARLop> <maskedAddrReg> "," <vectorSrc>

    <vp2-ARLop>            ::= "ARL" | "ARLC"
                             | "ARR" | "ARRC"

    <VECTORop-instruction> ::= <VECTORop> <maskedDstReg> "," <vectorSrc>

    <VECTORop>             ::= "LIT"
                             | "MOV"
                             | <vp11-VECTORop>
                             | <vp2-VECTORop>

    <vp11-VECTORop>        ::= "ABS"

    <vp2-VECTORop>         ::=         "ABSC"
                             | "FLR" | "FLRC"
                             | "FRC" | "FRCC"
                             |         "LITC"
                             |         "MOVC"
                             | "SSG" | "SSGC"

    <SCALARop-instruction> ::= <SCALARop> <maskedDstReg> "," <scalarSrc>

    <SCALARop>             ::= "EXP"
                             | "LOG"
                             | "RCP"
                             | "RSQ"
                             | <vp2-SCALARop>

    <vp2-SCALARop>         ::= "COS"  | "COSC"
                             | "EX2"  | "EX2C"
                             | "LG2"  | "LG2C"
                             |          "EXPC"
                             |          "LOGC"
                             |          "RCPC"
                             |          "RSQC"


    <BINop-instruction>    ::= <BINop> <maskedDstReg> "," <vectorSrc> ","
                               <vectorSrc>
    <BINop>                 ::= "ADD"
                             | "DP3"
                             | "DP4"
                             | "DST"
                             | "MAX"
                             | "MIN"
                             | "MUL"
                             | "SGE"
                             | "SLT"
                             | <vp11-BINop>
                             | <vp2-BINop>

    <vp11-BINop>            ::= "DPH"

    <vp2-BINop>             ::=         "ADDC"
                             |         "DP3C"
                             |         "DP4C"
                             |         "DPHC"
                             |         "DSTC"
                             |         "MAXC"
                             |         "MINC"
                             |         "MULC"
                             | "SEQ" | "SEQC"
                             | "SFL" | "SFLC"
                             |         "SGEC"
                             | "SGT" | "SGTC"
                             |         "SLTC"
                             | "SLE" | "SLEC"
                             | "SNE" | "SNEC"
                             | "STR" | "STRC"

    <TRIop-instruction>     ::= <TRIop> <maskedDstReg> "," <vectorSrc> ","
                               <vectorSrc> "," <vectorSrc>

    <TRIop>                 ::= "MAD"
                             | <vp2-TRIop>

    <vp2-TRIop>             ::= "MADC"

    <vp2-BRA-instruction>   ::= <vp2-BRANCHop> <vp2-branchLabel>
                                 <vp2-branchCondition>

    <vp2-BRANCHop>          ::= "BRA"
                             | "CAL"

    <vp2-RET-instruction>   ::= "RET" <vp2-branchCondition>

    <vp2-ARA-instruction>   ::= <vp2-ARAop> <maskedAddrReg> "," <addrRegister>

    <vp2-ARAop>             ::= "ARA" | "ARAC"

    <scalarSrc>             ::= <baseScalarSrc>
                             | <vp2-absScalarSrc>

    <vp2-absScalarSrc>      ::= <optionalSign> "|" <baseScalarSrc> "|"
    <baseScalarSrc>         ::= <optionalSign> <srcRegister> <scalarSuffix>

    <vectorSrc>             ::= <baseVectorSrc>
                              | <vp2-absVectorSrc>

    <vp2-absVectorSrc>      ::= <optionalSign> "|" <baseVectorSrc> "|"

    <baseVectorSrc>         ::= <optionalSign> <srcRegister> <swizzleSuffix>

    <srcRegister>           ::= <vtxAttribRegister>
                              | <progParamRegister>
                              | <tempRegister>

    <maskedDstReg>          ::= <dstRegister> <optionalWriteMask>
                                    <optionalCCMask>

    <dstRegister>           ::= <vtxResultRegister>
                              | <tempRegister>
                              | <vp2-nullRegister>

    <vp2-nullRegister>      ::= "CC"

    <vp2-branchCondition>   ::= <optionalCCMask>

    <vtxAttribRegister>     ::= "v" "[" vtxAttribRegNum "]"

    <vtxAttribRegNum>       ::= decimal integer from 0 to 15 inclusive
                              | "OPOS"
                              | "WGHT"
                              | "NRML"
                              | "COL0"
                              | "COL1"
                              | "FOGC"
                              | "TEX0"
                              | "TEX1"
                              | "TEX2"
                              | "TEX3"
                              | "TEX4"
                              | "TEX5"
                              | "TEX6"
                              | "TEX7"

    <progParamRegister>     ::= <absProgParamReg>
                              | <relProgParamReg>

    <absProgParamReg>       ::= "c" "[" <progParamRegNum> "]"

    <progParamRegNum>       ::= <vp1-progParamRegNum>
                              | <vp2-progParamRegNum>

    <vp1-progParamRegNum>   ::= decimal integer from 0 to 95 inclusive

    <vp2-progParamRegNum>   ::= decimal integer from 0 to 255 inclusive

    <relProgParamReg>       ::= "c" "[" <scalarAddr> <relProgParamOffset> "]"
    <relProgParamOffset>   ::= ""
                             | "+" <progParamPosOffset>
                             | "-" <progParamNegOffset>

    <progParamPosOffset>   ::= <vp1-progParamPosOff>
                             | <vp2-progParamPosOff>

    <vp1-progParamPosOff>  ::= decimal integer from 0 to 63 inclusive

    <vp2-progParamPosOff>  ::= decimal integer from 0 to 255 inclusive

    <progParamNegOffset>   ::= <vp1-progParamNegOff>
                             | <vp2-progParamNegOff>

    <vp1-progParamNegOff>  ::= decimal integer from 0 to 64 inclusive

    <vp2-progParamNegOff>  ::= decimal integer from 0 to 256 inclusive

    <tempRegister>         ::= "R0"  | "R1"  | "R2"  | "R3"
                             | "R4"  | "R5"  | "R6"  | "R7"
                             | "R8"  | "R9"  | "R10" | "R11"

    <vp2-tempRegister>     ::= "R12" | "R13" | "R14" | "R15"

    <vtxResultRegister>    ::= "o" "[" <vtxResultRegName> "]"

    <vtxResultRegName>     ::= "HPOS"
                             | "COL0"
                             | "COL1"
                             | "BFC0"
                             | "BFC1"
                             | "FOGC"
                             | "PSIZ"
                             | "TEX0"
                             | "TEX1"
                             | "TEX2"
                             | "TEX3"
                             | "TEX4"
                             | "TEX5"
                             | "TEX6"
                             | "TEX7"
                             | <vp2-resultRegName>

    <vp2-resultRegName>    ::= "CLP0"
                             | "CLP1"
                             | "CLP2"
                             | "CLP3"
                             | "CLP4"
                             | "CLP5"

    <scalarAddr>           ::= <addrRegister> "." <addrRegisterComp>

    <maskedAddrReg>        ::= <addrRegister> <addrWriteMask>

    <addrRegister>         ::= "A0"
                             | <vp2-addrRegister>
    <vp2-addrRegister>     ::= "A1"

    <addrRegisterComp>     ::= "x"
                             | <vp2-addrRegisterComp>

    <vp2-addrRegisterComp> ::= "y"
                             | "z"
                             | "w"

    <addrWriteMask>        ::= "." "x"
                             | <vp2-addrWriteMask>

    <vp2-addrWriteMask>     ::= ""
                             | "."     "y"
                             | "." "x" "y"
                             | "."         "z"
                             | "." "x"     "z"
                             | "."     "y" "z"
                             | "." "x" "y" "z"
                             | "."             "w"
                             | "." "x"         "w"
                             | "."     "y"     "w"
                             | "." "x" "y"     "w"
                             | "."         "z" "w"
                             | "." "x"     "z" "w"
                             | "."     "y" "z" "w"
                             | "." "x" "y" "z" "w"


    <optionalSign>         ::= ""
                             | "-"
                             | <vp2-optionalSign>

    <vp2-optionalSign>     ::= "+"

    <vp2-instructionLabel> ::= <vp2-branchLabel> ":"

    <vp2-branchLabel>      ::= <identifier>

    <optionalWriteMask>    ::= ""
                             | "." "x"
                             | "."     "y"
                             | "." "x" "y"
                             | "."         "z"
                             | "." "x"     "z"
                             | "."     "y" "z"
                             | "." "x" "y" "z"
                             | "."             "w"
                             | "." "x"         "w"
                             | "."     "y"     "w"
                             | "." "x" "y"     "w"
                             | "."         "z" "w"
                             | "." "x"     "z" "w"
                             | "."     "y" "z" "w"
                             | "." "x" "y" "z" "w"
```

```
<optionalCCMask>        ::= ""
                          | <vp2-ccMask>

<vp2-ccMask>            ::= "(" <vp2-ccMaskRule> <swizzleSuffix> ")"

<vp2-ccMaskRule>        ::= "EQ" | "GE" | "GT" | "LE" | "LT" | "NE"
                          | "TR" | "FL"

<scalarSuffix>          ::= "." <component>

<swizzleSuffix>         ::= ""
                          | "." <component>
                          | "." <component> <component>
                                <component> <component>

<component>             ::= "x"
                          | "y"
                          | "z"
                          | "w"
```

The <identifier> rule matches a sequence of one or more letters ("A" through "Z", "a" through "z", and "_") and digits ("0" through "9"); the first character must be a letter. The underscore ("_") counts as a letter. Upper and lower case letters are different (names are case-sensitive).

The <vertexAttribRegNum> rule matches both register numbers 0 through 15 and a set of mnemonics that abbreviate the aliasing of conventional per-vertex parameters to vertex attribute register numbers. Table X.3 shows the mapping from mnemonic to vertex attribute register number and what the mnemonic abbreviates.

Mnemonic	Vertex Attribute Register Number	Meaning
"OPOS"	0	object position
"WGHT"	1	vertex weight
"NRML"	2	normal
"COL0"	3	primary color
"COL1"	4	secondary color
"FOGC"	5	fog coordinate
"TEX0"	8	texture coordinate 0
"TEX1"	9	texture coordinate 1
"TEX2"	10	texture coordinate 2
"TEX3"	11	texture coordinate 3
"TEX4"	12	texture coordinate 4
"TEX5"	13	texture coordinate 5
"TEX6"	14	texture coordinate 6
"TEX7"	15	texture coordinate 7

Table X.3: The mapping between vertex attribute register numbers, mnemonics, and meanings.

A vertex program fails to load if it does not write at least one component of the HPOS register.

A vertex program fails to load in the VP1 execution environment if it contains more than 128 instructions. A vertex program fails to load in the VP2 execution environment if it contains more than 256 instructions. Each block of text matching the <instruction> rule counts as an instruction.

A vertex program fails to load if any instruction sources more than one unique program parameter register. An instruction can match the <progParamRegister> rule more than once only if all such matches are identical.

A vertex program fails to load if any instruction sources more than one unique vertex attribute register. An instruction can match the <vtxAttribRegister> rule more than once only if all such matches refer to the same register.

The error INVALID_OPERATION is generated if a vertex program fails to load because it is not syntactically correct or for one of the semantic restrictions listed above.

The error INVALID_OPERATION is generated if a program is loaded for id when id is currently loaded with a program of a different target.

A successfully loaded vertex program is parsed into a sequence of instructions. Each instruction is identified by its tokenized name. The operation of these instructions when executed is defined in section 2.14.1.10.

A successfully loaded program replaces the program previously assigned to the name specified by id. If the OUT_OF_MEMORY error is generated by LoadProgramNV, no change is made to the previous contents of the named program.

Querying the value of PROGRAM_ERROR_POSITION_NV returns a ubyte offset into the last loaded program string indicating where the first error in the program. If the program fails to load because of a semantic restriction that cannot be determined until the program is fully scanned, the error position will be len, the length of the program. If the program loads successfully, the value of PROGRAM_ERROR_POSITION_NV is assigned the value negative one.

Section 2.14.1.9, Vertex Program Binding and Program Management

The current vertex program is invoked whenever vertex attribute zero is updated (whether by a VertexAttributeNV or Vertex command). The current vertex program is updated by

```
  BindProgramNV(enum target, uint id);
```

where target must be VERTEX_PROGRAM_NV. This binds the vertex program named by id as the current vertex program. The error INVALID_OPERATION is generated if id names a program that is not a vertex program (for example, if id names a vertex state program as described in section 2.14.4).

Binding to a nonexistent program id does not generate an error. In particular, binding to program id zero does not generate an error.

However, because program zero cannot be loaded, program zero is always nonexistent. If a program id is successfully loaded with a new vertex program and id is also the currently bound vertex program, the new program is considered the currently bound vertex program.

The INVALID_OPERATION error is generated when both vertex program mode is enabled and Begin is called (or when a command that performs an implicit Begin is called) if the current vertex program is nonexistent or not valid. A vertex program may not be valid for reasons explained in section 2.14.5.

Programs are deleted by calling

```
  void DeleteProgramsNV(sizei n, const uint *ids);
```

ids contains n names of programs to be deleted. After a program is deleted, it becomes nonexistent, and its name is again unused. If a program that is currently bound is deleted, it is as though BindProgramNV has been executed with the same target as the deleted program and program zero. Unused names in ids are silently ignored, as is the value zero.

The command

```
  void GenProgramsNV(sizei n, uint *ids);
```

returns n previously unused program names in ids. These names are marked as used, for the purposes of GenProgramsNV only, but they become existent programs only when the are first loaded using LoadProgramNV. The error INVALID_VALUE is generated if n is negative.

An implementation may choose to establish a working set of programs on which binding and ExecuteProgramNV operations (execute programs are explained in section 2.14.4) are performed with higher performance. A program that is currently part of this working set is said to be resident.

The command

```
  boolean AreProgramsResidentNV(sizei n, const uint *ids,
                                boolean *residences);
```

returns TRUE if all of the n programs named in ids are resident, or if the implementation does not distinguish a working set. If at least one of the programs named in ids is not resident, then FALSE is returned, and the residence of each program is returned in residences. Otherwise the contents of residences are not changed. If any of the names in ids are nonexistent or zero, FALSE is returned, the error INVALID_VALUE is generated, and the contents of residences are indeterminate. The residence status of a single named program can also be queried by calling GetProgramivNV with id set to the name of the program and pname set to PROGRAM_RESIDENT_NV.

AreProgramsResidentNV indicates only whether a program is currently resident, not whether it could not be made resident. An implementation may choose to make a program resident only on first use, for example. The client may guide the GL implementation in determining which programs should be resident by requesting a set of programs to make resident.

The command

```
  void RequestResidentProgramsNV(sizei n, const uint *ids);
```

requests that the n programs named in ids should be made resident. While all the programs are not guaranteed to become resident, the implementation should make a best effort to make as many of the programs resident as possible. As a result of making the requested programs resident, program names not among the requested programs may become non-resident. Higher priority for residency should be given to programs listed earlier in the ids array. RequestResidentProgramsNV silently ignores attempts to make resident nonexistent program names or zero. AreProgramsResidentNV can be called after RequestResidentProgramsNV to determine which programs actually became resident.

Section 2.14.2, Vertex Program Operation

In the VP1 execution environment, there are twenty-one vertex program instructions. Four instructions (ABS, DPH, RCC, and SUB) are available only in the VP1.1 execution environment. The instructions and their respective input and output parameters are summarized in Table X.4.

Instruction	Inputs	Output	Description
ABS(*)	v	v	absolute value
ADD	v,v	v	add
ARL	v	as	address register load
DP3	v,v	ssss	3-component dot product
DP4	v,v	ssss	4-component dot product
DPH(*)	v,v	ssss	homogeneous dot product
DST	v,v	v	distance vector
EXP	s	v	exponential base 2 (approximate)
LIT	v	v	compute light coefficients
LOG	s	v	logarithm base 2 (approximate)
MAD	v,v,v	v	multiply and add
MAX	v,v	v	maximum
MIN	v,v	v	minimum
MOV	v	v	move
MUL	v,v	v	multiply
RCC(*)	s	ssss	reciprocal (clamped)
RCP	s	ssss	reciprocal
RSQ	s	ssss	reciprocal square root
SGE	v,v	v	set on greater than or equal
SLT	v,v	v	set on less than
SUB(*)	v,v	v	subtract

Table X.4: Summary of vertex program instructions in the VP1 execution environment. "v" indicates a floating-point vector input or output, "s" indicates a floating-point scalar input, "ssss" indicates a scalar output replicated across a 4-component vector, "as" indicates a single component of an address register.

In the VP2 execution environment, are thirty-nine vertex program instructions. Vertex program instructions may have an optional suffix of "C" to allow an update of the condition code register (section 2.14.1.6). For example, there are two instructions to perform vector addition, "ADD" and "ADDC". The vertex program instructions available in the VP2 execution environment and their respective input and output parameters are summarized in Table X.5.

Instruction	Inputs	Output	Description
ABS[C]	v	v	absolute value
ADD[C]	v,v	v	add
ARA[C]	av	av	address register add
ARL[C]	v	av	address register load
ARR[C]	v	av	address register load (with round)
BRA	as	none	branch
CAL	as	none	subroutine call
COS[C]	s	ssss	cosine
DP3[C]	v,v	ssss	3-component dot product
DP4[C]	v,v	ssss	4-component dot product
DPH[C]	v,v	ssss	homogeneous dot product
DST[C]	v,v	v	distance vector
EX2[C]	s	ssss	exponential base 2
EXP[C]	s	v	exponential base 2 (approximate)
FLR[C]	v	v	floor
FRC[C]	v	v	fraction
LG2[C]	s	ssss	logarithm base 2
LIT[C]	v	v	compute light coefficients
LOG[C]	s	v	logarithm base 2 (approximate)
MAD[C]	v,v,v	v	multiply and add
MAX[C]	v,v	v	maximum
MIN[C]	v,v	v	minimum
MOV[C]	v	v	move
MUL[C]	v,v	v	multiply
RCC[C]	s	ssss	reciprocal (clamped)
RCP[C]	s	ssss	reciprocal
RET	none	none	subroutine call return
RSQ[C]	s	ssss	reciprocal square root
SEQ[C]	v,v	v	set on equal
SFL[C]	v,v	v	set on false
SGE[C]	v,v	v	set on greater than or equal
SGT[C]	v,v	v	set on greater than
SIN[C]	s	ssss	sine
SLE[C]	v,v	v	set on less than or equal
SLT[C]	v,v	v	set on less than
SNE[C]	v,v	v	set on not equal
SSG[C]	v	v	set sign
STR[C]	v,v	v	set on true
SUB[C]	v,v	v	subtract

Table X.5: Summary of vertex program instructions in the VP2 execution environment. "v" indicates a floating-point vector input or output, "s" indicates a floating-point scalar input, "ssss" indicates a scalar output replicated across a 4-component vector, "av" indicates a full address register, "as" indicates a single component of an address register.

Section 2.14.2.1, Vertex Program Operands

Most vertex program instructions operate on floating-point vectors, floating-point scalars, or integer scalars as, indicated in the grammar (see section 2.14.1.8) by the rules <vectorSrc>, <scalarSrc>, and <scalarAddr>, respectively.

The basic set of floating-point scalar operands is defined by the grammar rule <baseScalarSrc>. Scalar operands are single components of vertex attribute, program parameter, or temporary registers, as allowed by the <srcRegister> rule. A vector component is selected by the <scalarSuffix> rule, where the characters "x", "y", "z", and "w" select the x, y, z, and w components, respectively, of the vector.

The basic set of floating-point vector operands is defined by the grammar rule <baseVectorSrc>. Vector operands can be obtained from vertex attribute, program parameter, or temporary registers as allowed by the <srcRegister> rule.

Basic vector operands can be swizzled according to the <swizzleSuffix> rule. In its most general form, the <swizzleSuffix> rule matches the pattern ".????" where each question mark is replaced with one of "x", "y", "z", or "w". For such patterns, the x, y, z, and w components of the operand are taken from the vector components named by the first, second, third, and fourth character of the pattern, respectively. For example, if the swizzle suffix is ".yzzx" and the specified source contains {2,8,9,0}, the swizzled operand used by the instruction is {8,9,9,2}.

If the <swizzleSuffix> rule matches "", it is treated as though it were ".xyzw". If the <swizzleSuffix> rule matches (ignoring whitespace) ".x", ".y", ".z", or ".w", these are treated the same as ".xxxx", ".yyyy", ".zzzz", and ".wwww" respectively.

Floating-point scalar or vector operands can optionally be negated according to the <negate> rules in <baseScalarSrc> and <baseVectorSrc>. If the <negate> matches "-", each operand or operand component is negated.

In the VP2 execution environment, a component-wise absolute value operation is performed on an operand if the <scalarSrc> or <vectorSrc> rules match <vp2-absScalarSrc> or <vp2-absVectorSrc>. In this case, the absolute value of each component of the operand is taken. In addition, if the <negate> rule in <vp2-absScalarSrc> or <vp2-absVectorSrc> matches "-", each component is subsequently negated.

Integer scalar operands are single components of one of the address register vectors, as identified by the <addrRegister> rule. A vector component is selected by the <scalarSuffix> rule in the same manner as floating-point scalar operands. Negation and absolute value operations are not available for integer scalar operands.

The following pseudo-code spells out the operand generation process. In the pseudo-code, "float" and "int" are floating-point and integer scalar types, while "floatVec" and "intVec" are four-component vectors. "source" is the register used for the operand, matching the <srcRegister> or <addrRegister> rules. "absolute" is TRUE if the operand matches the <vp2-absScalarSrc> or <vp2-absVectorSrc> rules, and FALSE otherwise. "negateBase" is TRUE if the <negate> rule in <baseScalarSrc> or

<baseVectorSrc> matches "-" and FALSE otherwise. "negateAbs" is TRUE if the <negate> rule in <vp2-absScalarSrc> or <vp2-absVectorSrc> matches "-" and FALSE otherwise. The ".c***", ".*c**", ".**c*", ".***c" modifiers refer to the x, y, z, and w components obtained by the swizzle operation.

```
  floatVec VectorLoad(floatVec source)
  {
      floatVec operand;

      operand.x = source.c***;
      operand.y = source.*c**;
      operand.z = source.**c*;
      operand.w = source.***c;
      if (negateBase) {
         operand.x = -operand.x;
         operand.y = -operand.y;
         operand.z = -operand.z;
         operand.w = -operand.w;
      }
      if (absolute) {
         operand.x = abs(operand.x);
         operand.y = abs(operand.y);
         operand.z = abs(operand.z);
         operand.w = abs(operand.w);
      }
      if (negateAbs) {
         operand.x = -operand.x;
         operand.y = -operand.y;
         operand.z = -operand.z;
         operand.w = -operand.w;
      }

      return operand;
  }

  float ScalarLoad(floatVec source)
  {
      float operand;

      operand = source.c***;
      if (negateBase) {
        operand = -operand;
      }
      if (absolute) {
         operand = abs(operand);
      }
      if (negateAbs) {
        operand = -operand;
      }

      return operand;
  }
  intVec AddrVectorLoad(intVec addrReg)
  {
      intVec operand;

      operand.x = source.c***;
      operand.y = source.*c**;
      operand.z = source.**c*;
      operand.w = source.***c;

      return operand;
  }

  int AddrScalarLoad(intVec addrReg)
  {
      return source.c***;
  }
```

If an operand is obtained from a program parameter register, by matching the <progParamRegister> rule, the register number can be obtained by absolute or relative addressing.

When absolute addressing is used, by matching the <absProgParamReg> rule, the program parameter register number is the number matching the <progParamRegNum>.

When relative addressing is used, by matching the <relProgParamReg> rule, the program parameter register number is computed during program execution. An index is computed by adding the integer scalar operand specified by the <scalarAddr> rule to the positive or negative offset specified by the <progParamOffset> rule. If <progParamOffset> matches "", an offset of zero is used.

The following pseudo-code spells out the process of loading a program parameter. "addrReg" refers to the address register used for relative addressing, "absolute" is TRUE if the operand uses absolute addressing and FALSE otherwise. "paramNumber" is the program parameter number for absolute addressing; "paramOffset" is the program parameter offset for relative addressing. "paramRegiser" is an array holding the complete set of program parameter registers.

```
  floatVec ProgramParameterLoad(intVec addrReg)
  {
    int index;

    if (absolute) {
      index = paramNumber;
    } else {
      index = AddrScalarLoad(addrReg) + paramOffset
    }

    return paramRegister[index];
  }
```

Section 2.14.2.2, Vertex Program Destination Register Update

Most vertex program instructions write a 4-component result vector to a single temporary, vertex result, or address register. Writes to individual components of the destination register are controlled by individual component write masks specified as part of the instruction. In the VP2 execution environment, writes are additionally controlled by the a condition code write mask, which is computed at run time.

The component write mask is specified by the <optionalWriteMask> rule found in the <maskedDstReg> or <maskedAddrReg> rule. If the optional mask is "", all components are enabled. Otherwise, the optional mask names the individual components to enable. The characters "x", "y", "z", and "w" match the x, y, z, and w components respectively. For example, an optional mask of ".xzw" indicates that the x, z, and w components should be enabled for writing but the y component should not. The grammar requires that the destination register mask components must be listed in "xyzw" order.

In the VP2 execution environment, the condition code write mask is specified by the <optionalCCMask> rule found in the <maskedDstReg> and <maskedAddrReg> rules. If the condition code mask matches "", all components are enabled. Otherwise, the condition code register is loaded and swizzled according to the swizzle codes specified by <swizzleSuffix>. Each component of the swizzled condition code is tested according to the rule given by <ccMaskRule>. <ccMaskRule> may have the values "EQ", "NE", "LT", "GE", LE", or "GT", which mean to enable writes if the corresponding condition code field evaluates to equal, not equal, less than, greater than or equal, less than or equal, or greater than, respectively. Comparisons involving condition codes of "UN" (unordered) evaluate to true for "NE" and false otherwise. For example, if the condition code is (GT,LT,EQ,GT) and the condition code mask is "(NE.zyxw)", the swizzle operation will load (EQ,LT,GT,GT) and the mask will thus will enable writes on the y, z, and w components. In addition, "TR" always enables writes and "FL" always disables writes, regardless of the condition code.

Each component of the destination register is updated with the result of the vertex program instruction if and only if the component is enabled for writes by the component write mask, and the optional condition code mask (if applicable). Otherwise, the component of the destination register remains unchanged.

In the VP2 execution environment, a vertex program instruction can also optionally update the condition code register. The condition code is updated if the condition code register update suffix "C" is present in the instruction. The instruction "ADDC" will update the condition code; the otherwise equivalent instruction "ADD" will not. If condition code updates are enabled, each component of the destination register enabled for writes is compared to zero. The corresponding component of the condition code is set to "LT", "EQ", or "GT", if the written component is less than, equal to, or greater than zero, respectively. Condition code components are set to "UN" if the written component is NaN. Values of -0.0 and +0.0 both evaluate to "EQ". If a component of the destination register is not enabled for writes, the corresponding condition code component is also unchanged.

In the following example code,

```
    # R1=(-2, 0, 2, NaN)              R0                  CC
    MOVC R0, R1;               # ( -2,  0,   2, NaN) (LT,EQ,GT,UN)
    MOVC R0.xyz, R1.yzwx;      # (  0,  2, NaN, NaN) (EQ,GT,UN,UN)
    MOVC R0 (NE), R1.zywx;     # (  0,  0, NaN,  -2) (EQ,EQ,UN,LT)
```

the first instruction writes (-2,0,2,NaN) to R0 and updates the condition code to (LT,EQ,GT,UN). The second instruction, only the "x", "y", and "z" components of R0 and the condition code are updated, so R0 ends up with (0,2,NaN,NaN) and the condition code ends up with (EQ,GT,UN,UN). In the third instruction, the condition code mask disables writes to the x component (its condition code field is "EQ"), so R0 ends up with (0,0,NaN,-2) and the condition code ends up with (EQ,EQ,UN,LT).
The following pseudocode illustrates the process of writing a result vector to the destination register. In the pseudocode, "instrmask" refers to the component write mask given by the <optionalWriteMask> rule. In the VP1 execution environment, "ccMaskRule" is always "" and "updatecc" is always FALSE. In the VP2 execution environment, "ccMaskRule" refers to the condition code mask rule given by <vp2-optionalCCMask> and "updatecc" is TRUE if and only if condition code updates are enabled. "result", "destination", and "cc" refer to the result vector, the register selected by <dstRegister> and the condition code, respectively. Condition codes do not exist in the VP1 execution environment.

```
  boolean TestCC(CondCode field) {
      switch (ccMaskRule) {
      case "EQ":  return (field == "EQ");
      case "NE":  return (field != "EQ");
      case "LT":  return (field == "LT");
      case "GE":  return (field == "GT" || field == "EQ");
      case "LE":  return (field == "LT" || field == "EQ");
      case "GT":  return (field == "GT");
      case "TR":  return TRUE;
      case "FL":  return FALSE;
      case "":    return TRUE;
      }
  }

  enum GenerateCC(float value) {
    if (value == NaN) {
      return UN;
    } else if (value < 0) {
      return LT;
    } else if (value == 0) {
      return EQ;
    } else {
      return GT;
    }
  }
  void UpdateDestination(floatVec destination, floatVec result)
  {
      floatVec merged;
      ccVec    mergedCC;

      // Merge the converted result into the destination register, under
      // control of the compile- and run-time write masks.
      merged = destination;
      mergedCC = cc;
      if (instrMask.x && TestCC(cc.c***)) {
          merged.x = result.x;
          if (updatecc) mergedCC.x = GenerateCC(result.x);
      }
      if (instrMask.y && TestCC(cc.*c**)) {
          merged.y = result.y;
          if (updatecc) mergedCC.y = GenerateCC(result.y);
      }
      if (instrMask.z && TestCC(cc.**c*)) {
          merged.z = result.z;
          if (updatecc) mergedCC.z = GenerateCC(result.z);
      }
      if (instrMask.w && TestCC(cc.***c)) {
          merged.w = result.w;
          if (updatecc) mergedCC.w = GenerateCC(result.w);
      }

      // Write out the new destination register and condition code.
      destination = merged;
      cc = mergedCC;
  }
```

Section 2.14.2.3, Vertex Program Execution

In the VP1 execution environment, vertex programs consist of a sequence of instructions without no support for branching. Vertex programs begin by executing the first instruction in the program, and execute instructions in the order specified in the program until the last instruction is reached.

VP2 vertex programs can contain one or more instruction labels, matching the grammar rule <vp2-instructionLabel>. An instruction label can be referred to explicitly in branch (BRA) or subroutine call (CAL) instructions. Instruction labels can be defined or used at any point in the body of a program, and can be used in instructions before being defined in the program string.

VP2 vertex program branching instructions can be conditional. The branch condition is specified by the <vp2-conditionMask> and may depend on the contents of the condition code register. Branch conditions are evaluated by evaluating a condition code write mask in exactly the same manner as done for register writes (section 2.14.2.2). If any of the four components of the condition code write mask are enabled, the branch is taken and execution continues with the instruction following the label specified in the instruction. Otherwise, the instruction is ignored and vertex program execution continues with the next instruction. In the following example code,

```
    MOVC CC, c[0];         # c[0]=(-2, 0, 2, NaN), CC gets (LT,EQ,GT,UN)
    BRA label1 (LT.xyzw);
    MOV R0,R1;             # not executed
  label1:
    BRA label2 (LT.wyzw);
    MOV R0,R2;             # executed
  label2:
```

the first BRA instruction loads a condition code of (LT,EQ,GT,UN) while the second BRA instruction loads a condition code of (UN,EQ,GT,UN). The first branch will be taken because the "x" component evaluates to LT; the second branch will not be taken because no component evaluates to LT.

VP2 vertex programs can specify subroutine calls. When a subroutine call (CAL) instruction is executed, a reference to the instruction immediately following the CAL instruction is pushed onto the call stack. When a subroutine return (RET) instruction is executed, an instruction reference is popped off the call stack and program execution continues with the popped instruction. A vertex program will terminate if a CAL instruction is executed with four entries already in the call stack or if a RET instruction is executed with an empty call stack.

If a VP2 vertex program has an instruction label "main", program execution begins with the instruction immediately following the instruction label. Otherwise, program execution begins with the first instruction of the program. Instructions will be executed sequentially in the order specified in the program, although branch instructions will affect the instruction execution order, as described above. A vertex program will terminate after executing a RET instruction with an empty call stack. A vertex program will also terminate after executing the last instruction in the program, unless that instruction was a taken branch.

A vertex program will fail to load if an instruction refers to a label that is not defined in the program string.

A vertex program will terminate abnormally if a subroutine call instruction produces a call stack overflow. Additionally, a vertex program will terminate abnormally after executing 65536 instructions to prevent hangs caused by infinite loops in the program.

When a vertex program terminates, normally or abnormally, it will emit a vertex whose attributes are taken from the final values of the vertex result registers (section 2.14.1.5).

Section 2.14.3, Vertex Program Instruction Set

The following sections describe the set of supported vertex program instructions. Instructions available only in the VP1.1 or VP2 execution environment will be noted in the instruction description.

Each section will contain pseudocode describing the instruction. Instructions will have up to three operands, referred to as "op0", "op1", and "op2". The operands are loaded using the mechanisms specified in section 2.14.2.1. Most instructions will generate a result vector called "result". The result vector is then written to the destination register specified in the instruction using the mechanisms specified in section 2.14.2.2.

Operands and results are represented as 32-bit single-precision floating-point numbers according to the IEEE 754 floating-point specification. IEEE denorm encodings, used to represent numbers smaller than 2^-126, are not supported. All such numbers are flushed to zero. There are three special encodings referred to in this section: +INF means "positive infinity", -INF means "negative infinity", and NaN refers to "not a number".

Arithmetic operations are typically carried out in single precision according to the rules specified in the IEEE 754 specification. Any exceptions and special cases will be noted in the instruction description.

Section 2.14.3.1, ABS: Absolute Value

The ABS instruction performs a component-wise absolute value operation on the single operand to yield a result vector.

```
  tmp = VectorLoad(op0);
  result.x = abs(tmp.x);
  result.y = abs(tmp.y);
  result.z = abs(tmp.z);
  result.w = abs(tmp.w);
```

The following special-case rules apply to absolute value operation:

1. abs(NaN) = NaN.
2. abs(-INF) = abs(+INF) = +INF.
3. abs(-0.0) = abs(+0.0) = +0.0.

The ABS instruction is available only in the VP1.1 and VP2 execution environments.

In the VP1.0 execution environment, the same functionality can be achieved with "MAX result, src, -src".

In the VP2 execution environment, the ABS instruction is effectively obsolete, since instructions can take the absolute value of each operand at no cost.

Section 2.14.3.2, ADD: Add

The ADD instruction performs a component-wise add of the two operands to yield a result vector.

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1);
result.x = tmp0.x + tmp1.x;
result.y = tmp0.y + tmp1.y;
result.z = tmp0.z + tmp1.z;
result.w = tmp0.w + tmp1.w;
```

The following special-case rules apply to addition:

```
1. "A+B" is always equivalent to "B+A".
2. NaN + <x> = NaN, for all <x>.
3. +INF + <x> = +INF, for all <x> except NaN and -INF.
4. -INF + <x> = -INF, for all <x> except NaN and +INF.
5. +INF + -INF = NaN.
6. -0.0 + <x> = <x>, for all <x>.
7. +0.0 + <x> = <x>, for all <x> except -0.0.
```

Section 2.14.3.3, ARA: Address Register Add

The ARA instruction adds two pairs of components of a vector address register operand to produce an integer result vector. The "x" and "z" components of the result vector contain the sum of the "x" and "z" components of the operand; the "y" and "w" components of the result vector contain the sum of the "y" and "w" components of the operand. Each component of the result vector is clamped to [-512, +511], the range of representable address register components.

```
itmp = AddrVectorLoad(op0);
iresult.x = itmp.x + itmp.z;
iresult.y = itmp.y + itmp.w;
iresult.z = itmp.x + itmp.z;
iresult.w = itmp.y + itmp.w;
if (iresult.x < -512) iresult.x = -512;
if (iresult.x > 511)  iresult.x = 511;
if (iresult.y < -512) iresult.y = -512;
if (iresult.y > 511)  iresult.y = 511;
if (iresult.z < -512) iresult.z = -512;
if (iresult.z > 511)  iresult.z = 511;
if (iresult.w < -512) iresult.w = -512;
if (iresult.w > 511)  iresult.w = 511;
```

Component swizzling is not supported when the operand is loaded.

The ARA instruction is available only in the VP2 execution environment.

Section 2.14.3.4, ARL: Address Register Load

In the VP1 execution environment, the ARL instruction loads a single scalar operand and performs a floor operation to generate an integer scalar to be written to the address register.

```
  tmp = ScalarLoad(op0);
  iresult.x = floor(tmp);
```

In the VP2 execution environment, the ARL instruction loads a single vector operand and performs a component-wise floor operation to generate an integer result vector. Each component of the result vector is clamped to [-512, +511], the range of representable address register components. The ARL instruction applies all masking operations to address register writes as are described in section 2.14.2.2.

```
  tmp = VectorLoad(op0);
  iresult.x = floor(tmp.x);
  iresult.y = floor(tmp.y);
  iresult.z = floor(tmp.z);
  iresult.w = floor(tmp.w);
  if (iresult.x < -512) iresult.x = -512;
  if (iresult.x > 511)  iresult.x = 511;
  if (iresult.y < -512) iresult.y = -512;
  if (iresult.y > 511)  iresult.y = 511;
  if (iresult.z < -512) iresult.z = -512;
  if (iresult.z > 511)  iresult.z = 511;
  if (iresult.w < -512) iresult.w = -512;
  if (iresult.w > 511)  iresult.w = 511;
```

The following special-case rules apply to floor computation:

1. floor(NaN) = NaN.
2. floor(<x>) = <x>, for -0.0, +0.0, -INF, and +INF. In all cases, the sign of the result is equal to the sign of the operand.

Section 2.14.3.5, ARR: Address Register Load (with round)

The ARR instruction loads a single vector operand and performs a component-wise round operation to generate an integer result vector. Each component of the result vector is clamped to [-512, +511], the range of representable address register components. The ARR instruction applies all masking operations to address register writes as described in section 2.14.2.2.

```
  tmp = VectorLoad(op0);
  iresult.x = round(tmp.x);
  iresult.y = round(tmp.y);
  iresult.z = round(tmp.z);
  iresult.w = round(tmp.w);
  if (iresult.x < -512) iresult.x = -512;
  if (iresult.x > 511)  iresult.x = 511;
  if (iresult.y < -512) iresult.y = -512;
  if (iresult.y > 511)  iresult.y = 511;
  if (iresult.z < -512) iresult.z = -512;
  if (iresult.z > 511)  iresult.z = 511;
  if (iresult.w < -512) iresult.w = -512;
  if (iresult.w > 511)  iresult.w = 511;
```

The rounding function, round(x), returns the nearest integer to <x>. If the fractional portion of <x> is 0.5, round(x) selects the nearest even integer.

The ARR instruction is available only in the VP2 execution environment.

Section 2.14.3.6, BRA: Branch

The BRA instruction conditionally transfers control to the instruction following the label specified in the instruction. The following pseudocode describes the operation of the instruction:

```
  if (TestCC(cc.c***) || TestCC(cc.*c**) ||
      TestCC(cc.**c*) || TestCC(cc.***c)) {
    // continue execution at instruction following <branchLabel>
  } else {
    // do nothing
  }
```

In the pseudocode, <branchLabel> is the label specified in the instruction matching the <vp2-branchLabel> grammar rule.

The BRA instruction is available only in the VP2 execution environment.

Section 2.14.3.7, CAL: Subroutine Call

The CAL instruction conditionally transfers control to the instruction following the label specified in the instruction. It also pushes a reference to the instruction immediately following the CAL instruction onto the call stack, where execution will continue after executing the matching RET instruction. The following pseudocode describes the operation of the instruction:

```
if (TestCC(cc.c***) || TestCC(cc.*c**) ||
    TestCC(cc.**c*) || TestCC(cc.***c)) {
  if (callStackDepth >= 4) {
    // terminate vertex program
  } else {
    callStack[callStackDepth] = nextInstruction;
    callStackDepth++;
  }
  // continue execution at instruction following <branchLabel>
} else {
  // do nothing
}
```

In the pseudocode, <branchLabel> is the label specified in the instruction matching the <vp2-branchLabel> grammar rule, <callStackDepth> is the current depth of the call stack, <callStack> is an array holding the call stack, and <nextInstruction> is a reference to the instruction immediately following the present one in the program string.

The CAL instruction is available only in the VP2 execution environment.

Section 2.14.3.8, COS: Cosine

The COS instruction approximates the cosine of the angle specified by the scalar operand and replicates the approximation to all four components of the result vector. The angle is specified in radians and does not have to be in the range [0,2*PI].

```
  tmp = ScalarLoad(op0);
  result.x = ApproxCosine(tmp);
  result.y = ApproxCosine(tmp);
  result.z = ApproxCosine(tmp);
  result.w = ApproxCosine(tmp);
```

The approximation function ApproxCosine is accurate to at least 22 bits with an angle in the range [0,2*PI].

```
  | ApproxCosine(x) - cos(x) | < 1.0 / 2^22, if 0.0 <= x < 2.0 * PI.
```

The error in the approximation will typically increase with the absolute value of the angle when the angle falls outside the range [0,2*PI].

The following special-case rules apply to cosine approximation:

1. ApproxCosine(NaN) = NaN.
2. ApproxCosine(+/-INF) = NaN.
3. ApproxCosine(+/-0.0) = +1.0.

The COS instruction is available only in the VP2 execution environment.

Section 2.14.3.9, DP3: 3-component Dot Product

The DP3 instruction computes a three component dot product of the two operands (using the x, y, and z components) and replicates the dot product to all four components of the result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1):
  result.x = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
             (tmp0.z * tmp1.z);
  result.y = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
             (tmp0.z * tmp1.z);
  result.z = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
             (tmp0.z * tmp1.z);
  result.w = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
             (tmp0.z * tmp1.z);
```

Section 2.14.3.10, DP4: 4-component Dot Product

The DP4 instruction computes a four component dot product of the two operands and replicates the dot product to all four components of the result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1):
  result.x = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
             (tmp0.z * tmp1.z) + (tmp0.w * tmp1.w);
  result.y = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
             (tmp0.z * tmp1.z) + (tmp0.w * tmp1.w);
  result.z = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
             (tmp0.z * tmp1.z) + (tmp0.w * tmp1.w);
  result.w = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
             (tmp0.z * tmp1.z) + (tmp0.w * tmp1.w);
```

Section 2.14.3.11, DPH: Homogeneous Dot Product

The DPH instruction computes a four-component dot product of the two operands, except that the W component of the first operand is assumed to be 1.0. The instruction replicates the dot product to all four components of the result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1):
  result.x = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
             (tmp0.z * tmp1.z) + tmp1.w;
  result.y = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
             (tmp0.z * tmp1.z) + tmp1.w;
  result.z = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
             (tmp0.z * tmp1.z) + tmp1.w;
  result.w = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
             (tmp0.z * tmp1.z) + tmp1.w;
```

The DPH instruction is available only in the VP1.1 and VP2 execution environments.

Section 2.14.3.12, DST: Distance Vector

The DST instruction computes a distance vector from two specially-formatted operands. The first operand should be of the form [NA, d^2, d^2, NA] and the second operand should be of the form [NA, 1/d, NA, 1/d], where NA values are not relevant to the calculation and d is a vector length. If both vectors satisfy these conditions, the result vector will be of the form [1.0, d, d^2, 1/d].

The exact behavior is specified in the following pseudo-code:

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = 1.0;
  result.y = tmp0.y * tmp1.y;
  result.z = tmp0.z;
  result.w = tmp1.w;
```

Given an arbitrary vector, d^2 can be obtained using the DP3 instruction (using the same vector for both operands) and 1/d can be obtained from d^2 using the RSQ instruction.

This distance vector is useful for per-vertex light attenuation calculations: a DP3 operation using the distance vector and an attenuation constants vector as operands will yield the attenuation factor.

Section 2.14.3.13, EX2: Exponential Base 2

The EX2 instruction approximates 2 raised to the power of the scalar operand and replicates it to all four components of the result vector.

```
  tmp = ScalarLoad(op0);
  result.x = Approx2ToX(tmp);
  result.y = Approx2ToX(tmp);
  result.z = Approx2ToX(tmp);
  result.w = Approx2ToX(tmp);
```

The approximation function is accurate to at least 22 bits:

```
  | Approx2ToX(x) - 2^x | < 1.0 / 2^22, if 0.0 <= x < 1.0,
```

and, in general,

```
  | Approx2ToX(x) - 2^x | < (1.0 / 2^22) * (2^floor(x)).
```

The following special-case rules apply to exponential approximation:

```
  1. Approx2ToX(NaN) = NaN.
  2. Approx2ToX(-INF) = +0.0.
  3. Approx2ToX(+INF) = +INF.
  4. Approx2ToX(+/-0.0) = +1.0.
```

The EX2 instruction is available only in the VP2 execution environment.

Section 2.14.3.14, EXP: Exponential Base 2 (approximate)

The EXP instruction computes a rough approximation of 2 raised to the power of the scalar operand. The approximation is returned in the "z" component of the result vector. A vertex program can also use the "x" and "y" components of the result vector to generate a more accurate approximation by evaluating

```
    result.x * f(result.y),
```

where f(x) is a user-defined function that approximates 2^x over the domain [0.0, 1.0). The "w" component of the result vector is always 1.0.

The exact behavior is specified in the following pseudo-code:

```
  tmp = ScalarLoad(op0);
  result.x = 2^floor(tmp);
  result.y = tmp - floor(tmp);
  result.z = RoughApprox2ToX(tmp);
  result.w = 1.0;
```

The approximation function is accurate to at least 11 bits:

```
  | RoughApprox2ToX(x) - 2^x | < 1.0 / 2^11, if 0.0 <= x < 1.0,
```

and, in general,

```
  | RoughApprox2ToX(x) - 2^x | < (1.0 / 2^11) * (2^floor(x)).
```

The following special cases apply to the EXP instruction:

```
  1. RoughApprox2ToX(NaN) = NaN.
  2. RoughApprox2ToX(-INF) = +0.0.
  3. RoughApprox2ToX(+INF) = +INF.
  4. RoughApprox2ToX(+/-0.0) = +1.0.
```

The EXP instruction is present for compatibility with the original NV_vertex_program instruction set; it is recommended that applications using NV_vertex_program2 use the EX2 instruction instead.

Section 2.14.3.15, FLR: Floor

The FLR instruction performs a component-wise floor operation on the operand to generate a result vector. The floor of a value is defined as the largest integer less than or equal to the value. The floor of 2.3 is 2.0; the floor of -3.6 is -4.0.

```
  tmp = VectorLoad(op0);
  result.x = floor(tmp.x);
  result.y = floor(tmp.y);
  result.z = floor(tmp.z);
  result.w = floor(tmp.w);
```

The following special-case rules apply to floor computation:

1. floor(NaN) = NaN.
2. floor(<x>) = <x>, for -0.0, +0.0, -INF, and +INF. In all cases, the sign of the result is equal to the sign of the operand.

The FLR instruction is available only in the VP2 execution environment.

Section 2.14.3.16, FRC: Fraction

The FRC instruction extracts the fractional portion of each component of the operand to generate a result vector. The fractional portion of a component is defined as the result after subtracting off the floor of the component (see FLR), and is always in the range [0.00, 1.00).

For negative values, the fractional portion is NOT the number written to the right of the decimal point -- the fractional portion of -1.7 is not 0.7 -- it is 0.3. 0.3 is produced by subtracting the floor of -1.7 (-2.0) from -1.7.

```
  tmp = VectorLoad(op0);
  result.x = tmp.x - floor(tmp.x);
  result.y = tmp.y - floor(tmp.y);
  result.z = tmp.z - floor(tmp.z);
  result.w = tmp.w - floor(tmp.w);
```

The following special-case rules, which can be derived from the rules for FLR and ADD apply to fraction computation:

1. fraction(NaN) = NaN.
2. fraction(+/-INF) = NaN.
3. fraction(+/-0.0) = +0.0.

The FRC instruction is available only in the VP2 execution environment.

Section 2.14.3.17, LG2: Logarithm Base 2

The LG2 instruction approximates the base 2 logarithm of the scalar operand and replicates it to all four components of the result vector.

```
  tmp = ScalarLoad(op0);
  result.x = ApproxLog2(tmp);
  result.y = ApproxLog2(tmp);
  result.z = ApproxLog2(tmp);
  result.w = ApproxLog2(tmp);
```

The approximation function is accurate to at least 22 bits:

```
  | ApproxLog2(x) - log_2(x) | < 1.0 / 2^22.
```

The following special-case rules apply to logarithm approximation:

```
  1. ApproxLog2(NaN) = NaN.
  2. ApproxLog2(+INF) = +INF.
  3. ApproxLog2(+/-0.0) = -INF.
  4. ApproxLog2(x) = NaN, -INF < x < -0.0.
  5. ApproxLog2(-INF) = NaN.
```

The LG2 instruction is available only in the VP2 execution environment.

Section 2.14.3.18, LIT: Compute Light Coefficients

The LIT instruction accelerates per-vertex lighting by computing lighting coefficients for ambient, diffuse, and specular light contributions. The "x" component of the operand is assumed to hold a diffuse dot product (n dot VP_pli, as in the vertex lighting equations in Section 2.13.1). The "y" component of the operand is assumed to hold a specular dot product (n dot h_i). The "w" component of the operand is assumed to hold the specular exponent of the material (s_rm), and is clamped to the range (-128, +128) exclusive.

The "x" component of the result vector receives the value that should be multiplied by the ambient light/material product (always 1.0). The "y" component of the result vector receives the value that should be multiplied by the diffuse light/material product (n dot VP_pli). The "z" component of the result vector receives the value that should be multiplied by the specular light/material product (f_i * (n dot h_i) ^ s_rm). The "w" component of the result is the constant 1.0.

Negative diffuse and specular dot products are clamped to 0.0, as is done in the standard per-vertex lighting operations. In addition, if the diffuse dot product is zero or negative, the specular coefficient is forced to zero.

```
  tmp = VectorLoad(op0);
  if (t.x < 0) t.x = 0;
  if (t.y < 0) t.y = 0;
  if (t.w < -(128.0-epsilon)) t.w = -(128.0-epsilon);
  else if (t.w > 128-epsilon) t.w = 128-epsilon;
  result.x = 1.0;
  result.y = t.x;
  result.z = (t.x > 0) ? RoughApproxPower(t.y, t.w) : 0.0;
  result.w = 1.0;
```

The exponentiation approximation function is defined in terms of the base 2 exponentiation and logarithm approximation operations in the EXP and LOG instructions, including errors and the processing of any special cases. In particular,

```
  RoughApproxPower(a,b) = RoughApproxExp2(b * RoughApproxLog2(a)).
```

The following special-case rules, which can be derived from the rules in the LOG, MUL, and EXP instructions, apply to exponentiation:

```
  1. RoughApproxPower(NaN, <x>) = NaN,
  2. RoughApproxPower(<x>, <y>) = NaN, if x <= -0.0,
  3. RoughApproxPower(+/-0.0, <x>) = +0.0, if x > +0.0, or
                                     +INF, if x < -0.0,
  4. RoughApproxPower(+1.0, <x>) = +1.0, if x is not NaN,
  5. RoughApproxPower(+INF, <x>) = +INF, if x > +0.0, or
                                   +0.0, if x < -0.0,
  6. RoughApproxPower(<x>, +/-0.0) = +1.0, if x >= -0.0
  7. RoughApproxPower(<x>, +INF) = +0.0, if -0.0 <= x < +1.0,
                                   +INF, if x > +1.0,
  8. RoughApproxPower(<x>, +INF) = +INF, if -0.0 <= x < +1.0,
                                   +0.0, if x > +1.0,
  9. RoughApproxPower(<x>, +1.0) = <x>, if x >= +0.0, and
  10. RoughApproxPower(<x>, NaN) = NaN.
```

Section 2.14.3.19, LOG: Logarithm Base 2 (Approximate)

The LOG instruction computes a rough approximation of the base 2 logarithm of the absolute value of the scalar operand. The approximation is returned in the "z" component of the result vector. A vertex program can also use the "x" and "y" components of the result vector to generate a more accurate approximation by evaluating

```
    result.x + f(result.y),
```

where f(x) is a user-defined function that approximates 2^x over the domain [1.0, 2.0). The "w" component of the result vector is always 1.0.

The exact behavior is specified in the following pseudo-code:

```
  tmp = fabs(ScalarLoad(op0));
  result.x = floor(log2(tmp));
  result.y = tmp / (2^floor(log2(tmp)));
  result.z = RoughApproxLog2(tmp);
  result.w = 1.0;
```

The approximation function is accurate to at least 11 bits:

```
  | RoughApproxLog2(x) - log_2(x) | < 1.0 / 2^11.
```

The following special-case rules apply to the LOG instruction:

1. RoughApproxLog2(NaN) = NaN.
2. RoughApproxLog2(+INF) = +INF.
3. RoughApproxLog2(+0.0) = -INF.

The LOG instruction is present for compatibility with the original NV_vertex_program instruction set; it is recommended that applications using NV_vertex_program2 use the LG2 instruction instead.

Section 2.14.3.20, MAD: Multiply And Add

The MAD instruction performs a component-wise multiply of the first two operands, and then does a component-wise add of the product to the third operand to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  tmp2 = VectorLoad(op2);
  result.x = tmp0.x * tmp1.x + tmp2.x;
  result.y = tmp0.y * tmp1.y + tmp2.y;
  result.z = tmp0.z * tmp1.z + tmp2.z;
  result.w = tmp0.w * tmp1.w + tmp2.w;
```

All special case rules applicable to the ADD and MUL instructions apply to the individual components of the MAD operation as well.

Section 2.14.3.21, MAX: Maximum

The MAX instruction computes component-wise maximums of the values in the two operands to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = max(tmp0.x, tmp1.x);
  result.y = max(tmp0.y, tmp1.y);
  result.z = max(tmp0.z, tmp1.z);
  result.w = max(tmp0.w, tmp1.w);
```

The following special cases apply to the maximum operation:

1. max(A,B) is always equivalent to max(B,A).
2. max(NaN, <x>) == NaN, for all <x>.

Section 2.14.3.22, MIN: Minimum

The MIN instruction computes component-wise minimums of the values in the two operands to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = min(tmp0.x, tmp1.x);
  result.y = min(tmp0.y, tmp1.y);
  result.z = min(tmp0.z, tmp1.z);
  result.w = min(tmp0.w, tmp1.w);
```

The following special cases apply to the minimum operation:

1. min(A,B) is always equivalent to min(B,A).
2. min(NaN, <x>) == NaN, for all <x>.

Section 2.14.3.23, MOV: Move

The MOV instruction copies the value of the operand to yield a result vector.

```
  result = VectorLoad(op0);
```

Section 2.14.3.24, MUL: Multiply

The MUL instruction performs a component-wise multiply of the two operands to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = tmp0.x * tmp1.x;
  result.y = tmp0.y * tmp1.y;
  result.z = tmp0.z * tmp1.z;
  result.w = tmp0.w * tmp1.w;
```

The following special-case rules apply to multiplication:

1. "A*B" is always equivalent to "B*A".
2. NaN * <x> = NaN, for all <x>.
3. +/-0.0 * +/-INF = NaN.
4. +/-0.0 * <x> = +/-0.0, for all <x> except -INF, +INF, and NaN. The sign of the result is positive if the signs of the two operands match and negative otherwise.
5. +/-INF * <x> = +/-INF, for all <x> except -0.0, +0.0, and NaN. The sign of the result is positive if the signs of the two operands match and negative otherwise.
6. +1.0 * <x> = <x>, for all <x>.

Section 2.14.3.25, RCC: Reciprocal (Clamped)

The RCC instruction approximates the reciprocal of the scalar operand, clamps the result to one of two ranges, and replicates the clamped result to all four components of the result vector.

If the approximate reciprocal is greater than 0.0, the result is clamped to the range [2^-64, 2^+64]. If the approximate reciprocal is not greater than zero, the result is clamped to the range [-2^+64, -2^-64].

```
  tmp = ScalarLoad(op0);
  result.x = ClampApproxReciprocal(tmp);
  result.y = ClampApproxReciprocal(tmp);
  result.z = ClampApproxReciprocal(tmp);
  result.w = ClampApproxReciprocal(tmp);
```

The approximation function is accurate to at least 22 bits:

```
  | ClampApproxReciprocal(x) - (1/x) | < 1.0 / 2^22, if 1.0 <= x < 2.0.
```

The following special-case rules apply to reciprocation:

```
  1. ClampApproxReciprocal(NaN) = NaN.
  2. ClampApproxReciprocal(+INF) = +2^-64.
  3. ClampApproxReciprocal(-INF) = -2^-64.
  4. ClampApproxReciprocal(+0.0) = +2^64.
  5. ClampApproxReciprocal(-0.0) = -2^64.
  6. ClampApproxReciprocal(x) = +2^-64, if -2^64 < x < +INF.
  7. ClampApproxReciprocal(x) = -2^-64, if -INF < x < -2^-64.
  8. ClampApproxReciprocal(x) = +2^64, if +0.0 < x < +2^-64.
  9. ClampApproxReciprocal(x) = -2^64, if -2^-64 < x < -0.0.
```

The RCC instruction is available only in the VP1.1 and VP2 execution environments.

Section 2.14.3.26, RCP: Reciprocal

The RCP instruction approximates the reciprocal of the scalar operand and replicates it to all four components of the result vector.

```
  tmp = ScalarLoad(op0);
  result.x = ApproxReciprocal(tmp);
  result.y = ApproxReciprocal(tmp);
  result.z = ApproxReciprocal(tmp);
  result.w = ApproxReciprocal(tmp);
```

The approximation function is accurate to at least 22 bits:

```
  | ApproxReciprocal(x) - (1/x) | < 1.0 / 2^22, if 1.0 <= x < 2.0.
```

The following special-case rules apply to reciprocation:

```
  1. ApproxReciprocal(NaN) = NaN.
  2. ApproxReciprocal(+INF) = +0.0.
  3. ApproxReciprocal(-INF) = -0.0.
  4. ApproxReciprocal(+0.0) = +INF.
  5. ApproxReciprocal(-0.0) = -INF.
```

Section 2.14.3.27, RET: Subroutine Call Return

The RET instruction conditionally returns from a subroutine initiated by a CAL instruction by popping an instruction reference off the top of the call stack and transferring control to the referenced instruction. The following pseudocode describes the operation of the instruction:

```
  if (TestCC(cc.c***) || TestCC(cc.*c**) ||
      TestCC(cc.**c*) || TestCC(cc.***c)) {
    if (callStackDepth <= 0) {
      // terminate vertex program
    } else {
      callStackDepth--;
      instruction = callStack[callStackDepth];
    }

    // continue execution at <instruction>
  } else {
    // do nothing
  }
```

In the pseudocode, <callStackDepth> is the depth of the call stack, <callStack> is an array holding the call stack, and <instruction> is a reference to an instruction previously pushed onto the call stack.

The RET instruction is available only in the VP2 execution environment.

Section 2.14.3.28, RSQ: Reciprocal Square Root

The RSQ instruction approximates the reciprocal of the square root of the scalar operand and replicates it to all four components of the result vector.

```
  tmp = ScalarLoad(op0);
  result.x = ApproxRSQRT(tmp);
  result.y = ApproxRSQRT(tmp);
  result.z = ApproxRSQRT(tmp);
  result.w = ApproxRSQRT(tmp);
```

The approximation function is accurate to at least 22 bits:

```
  | ApproxRSQRT(x) - (1/x) | < 1.0 / 2^22, if 1.0 <= x < 4.0.
```

The following special-case rules apply to reciprocal square roots:

```
  1. ApproxRSQRT(NaN) = NaN.
  2. ApproxRSQRT(+INF) = +0.0.
  3. ApproxRSQRT(-INF) = NaN.
  4. ApproxRSQRT(+0.0) = +INF.
  5. ApproxRSQRT(-0.0) = -INF.
  6. ApproxRSQRT(x) = NaN, if -INF < x < -0.0.
```

Section 2.14.3.29, SEQ: Set on Equal

The SEQ instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operand is equal to that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x == tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y == tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z == tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w == tmp1.w) ? 1.0 : 0.0;
  if (tmp0.x is NaN or tmp1.x is NaN) result.x = NaN;
  if (tmp0.y is NaN or tmp1.y is NaN) result.y = NaN;
  if (tmp0.z is NaN or tmp1.z is NaN) result.z = NaN;
  if (tmp0.w is NaN or tmp1.w is NaN) result.w = NaN;
```

The following special-case rules apply to SEQ:

```
  1. (<x> == <y>) and (<y> == <x>) always produce the same result.
  1. (NaN == <x>) is FALSE for all <x>, including NaN.
  2. (+INF == +INF) and (-INF == -INF) are TRUE.
  3. (-0.0 == +0.0) and (+0.0 == -0.0) are TRUE.
```

The SEQ instruction is available only in the VP2 execution environment.

Section 2.14.3.30, SFL: Set on False

The SFL instruction is a degenerate case of the other "Set on" instructions that sets all components of the result vector to 0.0.

```
  result.x = 0.0;
  result.y = 0.0;
  result.z = 0.0;
  result.w = 0.0;
```

The SFL instruction is available only in the VP2 execution environment.

Section 2.14.3.31, SGE: Set on Greater Than or Equal

The SGE instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operands is greater than or equal that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x >= tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y >= tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z >= tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w >= tmp1.w) ? 1.0 : 0.0;
  if (tmp0.x is NaN or tmp1.x is NaN) result.x = NaN;
  if (tmp0.y is NaN or tmp1.y is NaN) result.y = NaN;
  if (tmp0.z is NaN or tmp1.z is NaN) result.z = NaN;
  if (tmp0.w is NaN or tmp1.w is NaN) result.w = NaN;
```

The following special-case rules apply to SGE:

1. (NaN >= <x>) and (<x> >= NaN) are FALSE for all <x>.
2. (+INF >= +INF) and (-INF >= -INF) are TRUE.
3. (-0.0 >= +0.0) and (+0.0 >= -0.0) are TRUE.

Section 2.14.3.32, SGT: Set on Greater Than

The SGT instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operands is greater than that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x > tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y > tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z > tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w > tmp1.w) ? 1.0 : 0.0;
  if (tmp0.x is NaN or tmp1.x is NaN) result.x = NaN;
  if (tmp0.y is NaN or tmp1.y is NaN) result.y = NaN;
  if (tmp0.z is NaN or tmp1.z is NaN) result.z = NaN;
  if (tmp0.w is NaN or tmp1.w is NaN) result.w = NaN;
```

The following special-case rules apply to SGT:

1. (NaN > <x>) and (<x> > NaN) are FALSE for all <x>.
2. (-0.0 > +0.0) and (+0.0 > -0.0) are FALSE.

The SGT instruction is available only in the VP2 execution environment.

Section 2.14.3.33, SIN: Sine

The SIN instruction approximates the sine of the angle specified by the scalar operand and replicates it to all four components of the result vector. The angle is specified in radians and does not have to be in the range [0,2*PI].

```
  tmp = ScalarLoad(op0);
  result.x = ApproxSine(tmp);
  result.y = ApproxSine(tmp);
  result.z = ApproxSine(tmp);
  result.w = ApproxSine(tmp);
```

The approximation function is accurate to at least 22 bits with an angle in the range [0,2*PI].

```
  | ApproxSine(x) - sin(x) | < 1.0 / 2^22, if 0.0 <= x < 2.0 * PI.
```

The error in the approximation will typically increase with the absolute value of the angle when the angle falls outside the range [0,2*PI].

The following special-case rules apply to cosine approximation:

1. ApproxSine(NaN) = NaN.
2. ApproxSine(+/-INF) = NaN.
3. ApproxSine(+/-0.0) = +/-0.0. The sign of the result is equal to the sign of the single operand.

The SIN instruction is available only in the VP2 execution environment.

Section 2.14.3.34, SLE: Set on Less Than or Equal

The SLE instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operand is less than or equal to that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x <= tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y <= tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z <= tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w <= tmp1.w) ? 1.0 : 0.0;
  if (tmp0.x is NaN or tmp1.x is NaN) result.x = NaN;
  if (tmp0.y is NaN or tmp1.y is NaN) result.y = NaN;
  if (tmp0.z is NaN or tmp1.z is NaN) result.z = NaN;
  if (tmp0.w is NaN or tmp1.w is NaN) result.w = NaN;
```

The following special-case rules apply to SLE:

1. (NaN <= <x>) and (<x> <= NaN) are FALSE for all <x>.
2. (+INF <= +INF) and (-INF <= -INF) are TRUE.
3. (-0.0 <= +0.0) and (+0.0 <= -0.0) are TRUE.

The SLE instruction is available only in the VP2 execution environment.

Section 2.14.3.35, SLT: Set on Less Than

The SLT instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operand is less than that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x < tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y < tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z < tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w < tmp1.w) ? 1.0 : 0.0;
  if (tmp0.x is NaN or tmp1.x is NaN) result.x = NaN;
  if (tmp0.y is NaN or tmp1.y is NaN) result.y = NaN;
  if (tmp0.z is NaN or tmp1.z is NaN) result.z = NaN;
  if (tmp0.w is NaN or tmp1.w is NaN) result.w = NaN;
```

The following special-case rules apply to SLT:

1. (NaN < <x>) and (<x> < NaN) are FALSE for all <x>.
2. (-0.0 < +0.0) and (+0.0 < -0.0) are FALSE.

Section 2.14.3.36, SNE: Set on Not Equal

The SNE instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operand is not equal to that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x != tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y != tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z != tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w != tmp1.w) ? 1.0 : 0.0;
  if (tmp0.x is NaN or tmp1.x is NaN) result.x = NaN;
  if (tmp0.y is NaN or tmp1.y is NaN) result.y = NaN;
  if (tmp0.z is NaN or tmp1.z is NaN) result.z = NaN;
  if (tmp0.w is NaN or tmp1.w is NaN) result.w = NaN;
```

The following special-case rules apply to SNE:

1. (<x> != <y>) and (<y> != <x>) always produce the same result.
2. (NaN != <x>) is TRUE for all <x>, including NaN.
3. (+INF != +INF) and (-INF != -INF) are FALSE.
4. (-0.0 != +0.0) and (+0.0 != -0.0) are TRUE.

The SNE instruction is available only in the VP2 execution environment.

Section 2.14.3.37, SSG: Set Sign

The SSG instruction generates a result vector containing the signs of each component of the single operand. Each component of the result vector is 1.0 if the corresponding component of the operand is greater than zero, 0.0 if the corresponding component of the operand is equal to zero, and -1.0 if the corresponding component of the operand is less than zero.

```
  tmp = VectorLoad(op0);
  result.x = SetSign(tmp.x);
  result.y = SetSign(tmp.y);
  result.z = SetSign(tmp.z);
  result.w = SetSign(tmp.w);
```

The following special-case rules apply to SSG:

```
  1. SetSign(NaN) = NaN.
  2. SetSign(-0.0) = SetSign(+0.0) = 0.0.
  3. SetSign(-INF) = -1.0.
  4. SetSign(+INF) = +1.0.
  5. SetSign(x) = -1.0, if -INF < x < -0.0.
  6. SetSign(x) = +1.0, if +0.0 < x < +INF.
```

The SSG instruction is available only in the VP2 execution environment.

Section 2.14.3.38, STR: Set on True

The STR instruction is a degenerate case of the other "Set on" instructions that sets all components of the result vector to 1.0.

```
  result.x = 1.0;
  result.y = 1.0;
  result.z = 1.0;
  result.w = 1.0;
```

The STR instruction is available only in the VP2 execution environment.

Section 2.14.3.39, SUB: Subtract

The SUB instruction performs a component-wise subtraction of the second operand from the first to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = tmp0.x - tmp1.x;
  result.y = tmp0.y - tmp1.y;
  result.z = tmp0.z - tmp1.z;
  result.w = tmp0.w - tmp1.w;
```

The SUB instruction is completely equivalent to an identical ADD instruction in which the negate operator on the second operand is reversed:

1. "SUB R0, R1, R2" is equivalent to "ADD R0, R1, -R2".
2. "SUB R0, R1, -R2" is equivalent to "ADD R0, R1, R2".
3. "SUB R0, R1, |R2|" is equivalent to "ADD R0, R1, -|R2|".
4. "SUB R0, R1, -|R2|" is equivalent to "ADD R0, R1, |R2|".

The SUB instruction is available only in the VP1.1 and VP2 execution environments.

2.14.4 Vertex Arrays for Vertex Attributes

Data for vertex attributes in vertex program mode may be specified using vertex array commands. The client may specify and enable any of sixteen vertex attribute arrays.

The vertex attribute arrays are ignored when vertex program mode is disabled. When vertex program mode is enabled, vertex attribute arrays are used.

The command

```
  void VertexAttribPointerNV(uint index, int size, enum type,
                             sizei stride, const void *pointer);
```

describes the locations and organizations of the sixteen vertex attribute arrays. index specifies the particular vertex attribute to be described. size indicates the number of values per vertex that are stored in the array; size must be one of 1, 2, 3, or 4. type specifies the data type of the values stored in the array. type must be one of SHORT, FLOAT, DOUBLE, or UNSIGNED_BYTE and these values correspond to the array types short, int, float, double, and ubyte respectively. The INVALID_OPERATION error is generated if type is UNSIGNED_BYTE and size is not 4. The INVALID_VALUE error is generated if index is greater than 15. The INVALID_VALUE error is generated if stride is negative.

The one, two, three, or four values in an array that correspond to a single vertex attribute comprise an array element. The values within each array element at stored sequentially in memory. If the stride is specified as zero, then array elements are stored sequentially as well. Otherwise points to the ith and (i+1)st elements of an array differ by stride basic machine units (typically unsigned bytes), the pointer to the (i+1)st element being greater. pointer specifies the location in memory of the first value of the first element of the array being specified.

Vertex attribute arrays are enabled with the EnableClientState command and disabled with the DisableClientState command. The value of the argument to either command is VERTEX_ATTRIB_ARRAYi_NV where i is an integer between 0 and 15; specifying a value of i enables or disables the vertex attribute array with index i. The constants obey VERTEX_ATTRIB_ARRAYi_NV = VERTEX_ATTRIB_ARRAY0_NV + i.

When vertex program mode is enabled, the ArrayElement command operates as described in this section in contrast to the behavior described in section 2.8. Likewise, any vertex array transfer commands that are defined in terms of ArrayElement (DrawArrays, DrawElements, and DrawRangeElements) assume the operation of ArrayElement described in this section when vertex program mode is enabled.

When vertex program mode is enabled, the ArrayElement command transfers the ith element of particular enabled vertex arrays as described below. For each enabled vertex attribute array, it is as though the corresponding command from section 2.14.1.1 were called with a pointer to element i. For each vertex attribute, the corresponding command is VertexAttrib[size][type]v, where size is one of [1,2,3,4], and type is one of [s,f,d,ub], corresponding to the array types short, int, float, double, and ubyte respectively.

However, if a given vertex attribute array is disabled, but its corresponding aliased conventional per-vertex parameter's vertex array (as described in section 2.14.1.6) is enabled, then it is as though the corresponding command from section 2.7 or section 2.6.2 were called with a pointer to element i. In this case, the corresponding command is determined as described in section 2.8's description of ArrayElement.

If the vertex attribute array 0 is enabled, it is as though VertexAttrib[size][type]v(0, ...) is executed last, after the executions of other corresponding commands. If the vertex attribute array 0 is disabled but the vertex array is enabled, it is as though Vertex[size][type]v is executed last, after the executions of other corresponding commands.

2.14.5 Vertex State Programs

Vertex state programs share the same instruction set as and a similar execution model to vertex programs. While vertex programs are executed implicitly when a vertex transformation is provoked, vertex state programs are executed explicitly, independently of any vertices. Vertex state programs can write program parameter registers, but may not write vertex result registers. Vertex state programs have not been extended beyond the the VP1.0 execution environment, and are offered solely for compatibility with that execution environment.

The purpose of a vertex state program is to update program parameter registers by means of an application-defined program. Typically, an application will load a set of program parameters and then execute a vertex state program that reads and updates the program parameter

```
    registers.  For example, a vertex state program might normalize a set of
    unnormalized vectors previously loaded as program parameters.  The
    expectation is that subsequently executed vertex programs would use the
    normalized program parameters.

    Vertex state programs are loaded with the same LoadProgramNV command (see
    section 2.14.1.8) used to load vertex programs except that the target must
    be VERTEX_STATE_PROGRAM_NV when loading a vertex state program.

    Vertex state programs must conform to a more limited grammar than the
    grammar for vertex programs.  The vertex state program grammar for
    syntactically valid sequences is the same as the grammar defined in
    section 2.14.1.8 with the following modified rules:

    <program>              ::= <vp1-program>

    <vp1-program>          ::= "!!VSP1.0" <programBody> "END"

    <dstReg>               ::= <absProgParamReg>
                             | <temporaryReg>

    <vertexAttribReg>      ::= "v" "[" "0" "]"

    A vertex state program fails to load if it does not write at least
    one program parameter register.

    A vertex state program fails to load if it contains more than 128
    instructions.

    A vertex state program fails to load if any instruction sources more
    than one unique program parameter register.

    A vertex state program fails to load if any instruction sources
    more than one unique vertex attribute register (this is necessarily
    true because only vertex attribute 0 is available in vertex state
    programs).

    The error INVALID_OPERATION is generated if a vertex state program
    fails to load because it is not syntactically correct or for one
    of the other reasons listed above.

    A successfully loaded vertex state program is parsed into a sequence
    of instructions.  Each instruction is identified by its tokenized
    name.  The operation of these instructions when executed is defined
    in section 2.14.1.10.

    Executing vertex state programs is legal only outside a Begin/End
    pair.  A vertex state program may not read any vertex attribute
    register other than register zero.  A vertex state program may not
    write any vertex result register.

    The command

      ExecuteProgramNV(enum target, uint id, const float *params);

    executes the vertex state program named by id.  The target must be
    VERTEX_STATE_PROGRAM_NV and the id must be the name of program loaded
```

with a target type of VERTEX_STATE_PROGRAM_NV. params points to an array of four floating-point values that are loaded into vertex attribute register zero (the only vertex attribute readable from a vertex state program).

The INVALID_OPERATION error is generated if the named program is nonexistent, is invalid, or the program is not a vertex state program. A vertex state program may not be valid for reasons explained in section 2.14.5.

2.14.6, Program Options

In the VP1.1 and VP2.0 execution environment, vertex programs may specify one or more program options that modify the execution environment, according to the <option> grammar rule. The set of options available to the program is described below.

Section 2.14.6.1, Position-Invariant Vertex Program Option

If <vp11-option> or <vp2-option> matches "NV_position_invariant", the vertex program is presumed to be position-invariant. By default, vertex programs are not position-invariant. Even if programs emulate the conventional OpenGL transformation model, they may still not produce the exact same transform results, due to rounding errors or different operation orders. Such programs may not work well for multi-pass rendering algorithms where the second and subsequent passes use an EQUAL depth test.

Position-invariant vertex programs do not compute a final vertex position; instead, the GL computes vertex coordinates as described in section 2.10. This computation should produce exactly the same results as the conventional OpenGL transformation model, assuming vertex weighting and vertex blending are disabled.

A vertex program that specifies the position-invariant option will fail to load if it writes to the HPOS result register.

Additionally, in the VP1.1 execution environment, position-invariant programs can not use relative addressing for program parameters. Any position-invariant VP1.1 program matches the grammar rule <relProgParamReg>, will fail to load. No such restriction exists for VP2.0 programs.

For position-invariant programs, the limit on the number of instructions allowed in a program is reduced by four: position-invariant VP1.1 and VP2.0 programs may have no more than 124 or 252 instructions, respectively.

2.14.7 Tracking Matrices

As a convenience to applications, standard GL matrix state can be tracked into program parameter vectors. This permits vertex programs to access matrices specified through GL matrix commands.

In addition to GL's conventional matrices, several additional matrices are available for tracking. These matrices have names of the form MATRIXi_NV where i is between zero and n-1 where n is the value

of the MAX_TRACK_MATRICES_NV implementation dependent constant.
The MATRIXi_NV constants obey MATRIXi_NV = MATRIX0_NV + i. The value
of MAX_TRACK_MATRICES_NV must be at least eight. The maximum
stack depth for tracking matrices is defined by the
MAX_TRACK_MATRIX_STACK_DEPTH_NV and must be at least 1.

The command

```
  TrackMatrixNV(enum target, uint address, enum matrix, enum transform);
```

tracks a given transformed version of a particular matrix into
a contiguous sequence of four vertex program parameter registers
beginning at address. target must be VERTEX_PROGRAM_NV (though
tracked matrices apply to vertex state programs as well because both
vertex state programs and vertex programs shared the same program
parameter registers). matrix must be one of NONE, MODELVIEW,
PROJECTION, TEXTURE, TEXTUREi_ARB (where i is between 0 and n-1
where n is the number of texture units supported), COLOR (if
the ARB_imaging subset is supported), MODELVIEW_PROJECTION_NV,
or MATRIXi_NV. transform must be one of IDENTITY_NV, INVERSE_NV,
TRANSPOSE_NV, or INVERSE_TRANSPOSE_NV. The INVALID_VALUE error is
generated if address is not a multiple of four.

The MODELVIEW_PROJECTION_NV matrix represents the concatenation of
the current modelview and projection matrices. If M is the current
modelview matrix and P is the current projection matrix, then the
MODELVIEW_PROJECTION_NV matrix is C and computed as

```
    C = P M
```

Matrix tracking for the specified program parameter register and the
next consecutive three registers is disabled when NONE is supplied
for matrix. When tracking is disabled the previously tracked program
parameter registers retain the state of their last tracked values.
Otherwise, the specified transformed version of matrix is tracked into
the specified program parameter register and the next three registers.
Whenever the matrix changes, the transformed version of the matrix
is updated in the specified range of program parameter registers.
If TEXTURE is specified for matrix, the texture matrix for the current
active texture unit is tracked. If TEXTUREi_ARB is specified for
matrix, the <i>th texture matrix is tracked.

Matrices are tracked row-wise meaning that the top row of the
transformed matrix is loaded into the program parameter address,
the second from the top row of the transformed matrix is loaded into
the program parameter address+1, the third from the top row of the
transformed matrix is loaded into the program parameter address+2,
and the bottom row of the transformed matrix is loaded into the
program parameter address+3. The transformed matrix may be identical
to the specified matrix, the inverse of the specified matrix, the
transpose of the specified matrix, or the inverse transpose of the
specified matrix, depending on the value of transform.

When matrix tracking is enabled for a particular program parameter
register sequence, updates to the program parameter using
ProgramParameterNV commands, a vertex program, or a vertex state
program are not possible. The INVALID_OPERATION error is generated

if a ProgramParameterNV command is used to update a program parameter register currently tracking a matrix.

The INVALID_OPERATION error is generated by ExecuteProgramNV when the vertex state program requested for execution writes to a program parameter register that is currently tracking a matrix because the program is considered invalid.

2.14.8 Required Vertex Program State

The state required for vertex programs consists of:

a bit indicating whether or not program mode is enabled;

a bit indicating whether or not two-sided color mode is enabled;

a bit indicating whether or not program-specified point size mode is enabled;

256 4-component floating-point program parameter registers;

16 4-component vertex attribute registers (though this state is aliased with the current normal, primary color, secondary color, fog coordinate, weights, and texture coordinate sets);

24 sets of matrix tracking state for each set of four sequential program parameter registers, consisting of a n-valued integer indicated the tracked matrix or GL_NONE (where n is 5 + the number of texture units supported + the number of tracking matrices supported) and a four-valued integer indicating the transformation of the tracked matrix;

an unsigned integer naming the currently bound vertex program

and the state must be maintained to indicate which integers are currently in use as program names.

Each existent program object consists of a target, a boolean indicating whether the program is resident, an array of type ubyte containing the program string, and the length of the program string array. Initially, no program objects exist.

Program mode, two-sided color mode, and program-specified point size mode are all initially disabled.

The initial state of all 256 program parameter registers is (0,0,0,0).

The initial state of the 16 vertex attribute registers is (0,0,0,1) except in cases where a vertex attribute register aliases to a conventional GL transform mode vertex parameter in which case the initial state is the initial state of the respective aliased conventional vertex parameter.

The initial state of the 24 sets of matrix tracking state is NONE for the tracked matrix and IDENTITY_NV for the transformation of the tracked matrix.

The initial currently bound program is zero.

The client state required to implement the 16 vertex attribute arrays consists of 16 boolean values, 16 memory pointers, 16 integer stride values, 16 symbolic constants representing array types, and 16 integers representing values per element. Initially, the boolean values are each disabled, the memory pointers are each null, the strides are each zero, the array types are each FLOAT, and the integers representing values per element are each four."

Additions to Chapter 3 of the OpenGL 1.3 Specification (Rasterization)

None.

Additions to Chapter 4 of the OpenGL 1.3 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 1.3 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 1.3 Specification (State and State Requests)

None.

Additions to Appendix A of the OpenGL 1.3 Specification (Invariance)

None.

Additions to the AGL/GLX/WGL Specifications

None.

GLX Protocol

All relevant protocol is defined in the NV_vertex_program extension.

Errors

This list includes the errors specified in the NV_vertex_program extension, modified as appropriate.

The error INVALID_VALUE is generated if VertexAttribNV is called where index is greater than 15.

The error INVALID_VALUE is generated if any ProgramParameterNV has an index is greater than 255 (was 95 in NV_vertex_program).

The error INVALID_VALUE is generated if VertexAttribPointerNV is called where index is greater than 15.

The error INVALID_VALUE is generated if VertexAttribPointerNV is called where size is not one of 1, 2, 3, or 4.

The error INVALID_VALUE is generated if VertexAttribPointerNV is called where stride is negative.

The error INVALID_OPERATION is generated if VertexAttribPointerNV is called where type is UNSIGNED_BYTE and size is not 4.

The error INVALID_VALUE is generated if LoadProgramNV is used to load a program with an id of zero.

The error INVALID_OPERATION is generated if LoadProgramNV is used to load an id that is currently loaded with a program of a different program target.

The error INVALID_OPERATION is generated if the program passed to LoadProgramNV fails to load because it is not syntactically correct based on the specified target. The value of PROGRAM_ERROR_POSITION_NV is still updated when this error is generated.

The error INVALID_OPERATION is generated if LoadProgramNV has a target of VERTEX_PROGRAM_NV and the specified program fails to load because it does not write the HPOS register at least once. The value of PROGRAM_ERROR_POSITION_NV is still updated when this error is generated.

The error INVALID_OPERATION is generated if LoadProgramNV has a target of VERTEX_STATE_PROGRAM_NV and the specified program fails to load because it does not write at least one program parameter register. The value of PROGRAM_ERROR_POSITION_NV is still updated when this error is generated.

The error INVALID_OPERATION is generated if the vertex program or vertex state program passed to LoadProgramNV fails to load because it contains more than 128 instructions (VP1 programs) or 256 instructions (VP2 programs). The value of PROGRAM_ERROR_POSITION_NV is still updated when this error is generated.

The error INVALID_OPERATION is generated if a program is loaded with LoadProgramNV for id when id is currently loaded with a program of a different target.

The error INVALID_OPERATION is generated if BindProgramNV attempts to bind to a program name that is not a vertex program (for example, if the program is a vertex state program).

The error INVALID_VALUE is generated if GenProgramsNV is called where n is negative.

The error INVALID_VALUE is generated if AreProgramsResidentNV is called and any of the queried programs are zero or do not exist.

The error INVALID_OPERATION is generated if ExecuteProgramNV executes a program that does not exist.

The error INVALID_OPERATION is generated if ExecuteProgramNV executes a program that is not a vertex state program.

```
    The error INVALID_OPERATION is generated if Begin, RasterPos, or a command
    that performs an explicit Begin is called when vertex program mode is
    enabled and the currently bound vertex program writes program parameters
    that are currently being tracked.

    The error INVALID_OPERATION is generated if ExecuteProgramNV is called and
    the vertex state program to execute writes program parameters that are
    currently being tracked.

    The error INVALID_VALUE is generated if TrackMatrixNV has a target of
    VERTEX_PROGRAM_NV and attempts to track an address is not a multiple of
    four.

    The error INVALID_VALUE is generated if GetProgramParameterNV is called to
    query an index greater than 255 (was 95 in NV_vertex_program).

    The error INVALID_VALUE is generated if GetVertexAttribNV is called to
    query an <index> greater than 15, or if <index> is zero and <pname> is
    CURRENT_ATTRIB_NV.

    The error INVALID_VALUE is generated if GetVertexAttribPointervNV is
    called to query an index greater than 15.

    The error INVALID_OPERATION is generated if GetProgramivNV is called and
    the program named id does not exist.

    The error INVALID_OPERATION is generated if GetProgramStringNV is called
    and the program named <program> does not exist.

    The error INVALID_VALUE is generated if GetTrackMatrixivNV is called with
    an <address> that is not divisible by four or greater than or equal to 256
    (was 96 in NV_vertex_program).

    The error INVALID_VALUE is generated if AreProgramsResidentNV,
    DeleteProgramsNV, GenProgramsNV, or RequestResidentProgramsNV are called
    where <n> is negative.

    The error INVALID_VALUE is generated if LoadProgramNV is called where
    <len> is negative.

    The error INVALID_VALUE is generated if ProgramParameters4dvNV or
    ProgramParameters4fvNV are called where <count> is negative.

    The error INVALID_VALUE is generated if VertexAttribs{1,2,3,4}{d,f,s}vNV
    is called where <count> is negative.

    The error INVALID_ENUM is generated if BindProgramNV,
    GetProgramParameterfvNV, GetProgramParameterdvNV, GetTrackMatrixivNV,
    ProgramParameter4fNV, ProgramParameter4dNV, ProgramParameter4fvNV,
    ProgramParameter4dvNV, ProgramParameters4fvNV, ProgramParameters4dvNV,
    or TrackMatrixNV are called where <target> is not VERTEX_PROGRAM_NV.

    The error INVALID_ENUM is generated if LoadProgramNV or
    ExecuteProgramNV are called where <target> is not either
    VERTEX_PROGRAM_NV or VERTEX_STATE_PROGRAM_NV.
```

New State

(Modify Table X.5, New State Introduced by NV_vertex_program from the NV_vertex_program specification.)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
PROGRAM_PARAMETER_NV	256xR4	GetProgramParameterNV	(0,0,0,0)	program parameters	2.14.1.2	-

(Modify Table X.7. Vertex Program Per-vertex Execution State. "VP1" and "VP2" refer to the VP1 and VP2 execution environments, respectively.)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
-	12xR4	-	(0,0,0,0)	VP1 temporary registers	2.14.1.4	-
-	16xR4	-	(0,0,0,0)	VP2 temporary registers	2.14.1.4	-
-	15xR4	-	(0,0,0,1)	vertex result registers	2.14.1.4	-
	Z4	-	(0,0,0,0)	VP1 address register	2.14.1.3	-
	2xZ4	-	(0,0,0,0)	VP2 address registers	2.14.1.3	-

Name

    NV_vertex_program2_option

Name Strings

    GL_NV_vertex_program2_option

Status

    Shipping.

Version

```
    Last Modified:      06/23/2004
    NVIDIA Revision:    3
```

Number

    305

Dependencies

    ARB_vertex_program is required.

Overview

```
    This extension provides additional vertex program functionality
    to extend the standard ARB_vertex_program language and execution
    environment.  ARB programs wishing to use this added functionality
    need only add:

        OPTION NV_vertex_program2;

    to the beginning of their vertex programs.

    The functionality provided by this extension, which is roughly
    equivalent to that provided by the NV_vertex_program2 extension,
    includes:

      * general purpose dynamic branching,

      * subroutine calls,

      * data-dependent conditional write masks,

      * programmable user clip distances,

      * address registers with four components (instead of just one),

      * absolute value operator on scalar and swizzled operand loads,

      * rudimentary address register math,

      * SIN and COS trigonometry instructions, and

      * fully orthogonal "set on" instructions, including a "set sign"
```

        instruction.

Issues

Why is this a separate extension, rather than just an additional feature of NV_vertex_program2?

RESOLVED: The NV_vertex_program2 specification was completed (with a published implementation) prior to the completion of ARB_vertex_program. Future NVIDIA vertex program extensions should contain extensions to the ARB_vertex_program execution environment as a standard feature.

NV_vertex_program1_1 contains one feature not found in ARB_vertex_program: the "RCC" (reciprocal clamped) instruction. Should a "NV_vertex_program1_1" program option be provided to expose this small amount of missing functionality?

RESOLVED: No. By itself, that functionality is not all that interesting.

Should this extension provide a mechanism to specify an "ARB" version of NV_vertex_program state programs (!!VSP1.0)?

RESOLVED: No.

Should a similar option be provided to expose ARB_vertex_program features not found in NV_vertex_program (e.g., local parameters, state bindings, certain "macro" instructions) under the NV_vertex_program interface?

RESOLVED: No. Why not just write an ARB program in that case?

The ARB_vertex_program spec has a minor grammar bug that requires that inline scalar constants used as scalar operands include a component selector. In other words, you have to say "11.0.x" to use the constant "11.0". What should we do here?

RESOLVED: The NV_vertex_program2_option grammar will correct this problem, which should be fixed in future revisions to the ARB language.

New Procedures and Functions

None.

New Tokens

Accepted by the <pname> parameter of GetProgramivARB:

```
    MAX_PROGRAM_EXEC_INSTRUCTIONS_NV                0x88F4
    MAX_PROGRAM_CALL_DEPTH_NV                       0x88F5
```

Additions to Chapter 2 of the OpenGL 1.4 Specification (OpenGL Operation)

Modify Section 2.11, Clipping (p. 42)

(insert before the second paragraph, p. 43) In vertex program mode, conventional user clipping is performed if the vertex program is position-invariant (section 2.14.4.5.1). When the vertex program is not position-invariant, it can write a single floating-point clip distance for each supported clip plane. The half-space corresponding to clip plane <n> is given by the set of points that satisfy the inequality

```
  c_n(P) >=0,
```

where c_n(P) is the value of clip distance <n> at point P. For point primitives, c_n(P) is simply the clip distance for the vertex in question. For line and triangle primitives, per-vertex clip distances are interpolated using a weighted mean, with weights derived according to the algorithms described in sections 3.4 and 3.5.

Modify Section 2.14.2, Vertex Program Grammar and Restrictions

(mostly add to existing grammar rules, modify a few existing grammar rules -- changes marked with "***")

```
<optionName>            ::= "NV_vertex_program2"

<statement>             ::= <branchLabel> ":"

<instruction>           ::= <FlowInstruction>

<ALUInstruction>        ::= <ARAop_instruction>

<FlowInstruction>       ::= <BRAop_instruction>
                          | <FLOWCCop_instruction>

<VECTORop>              ::= "SSG"

<SCALARop>              ::= "COS"
                          | "RCC"
                          | "SIN"

<BINop>                 ::= "SEQ"
                          | "SFL"
                          | "SGT"
                          | "SLE"
                          | "SNE"
                          | "STR"

<ARLop>                 ::= "ARR"

<ARLop_src>             ::= <instOperandV>
                              (*** instead of <instOperandS>)

<ARAop_instruction>     ::= <ARAop> <instResultAddr> ","
                            <instOperandAddrVNS>
```

```
    <ARAop>                  ::= "ARA"

    <BRAop_instruction>      ::= <BRAop> <branchLabel> <optBranchCond>

    <BRAop>                  ::= "BRA"
                               | "CAL"

    <FLOWCCop_instruction>   ::= <FLOWCCop> <optBranchCond>

    <FLOWCCop>               ::= "RET"

    <optBranchCond>          ::= /* empty */
                               | <ccMask>

    <instOperandV>           ::= <instOperandAbsV>

    <instOperandAbsV>        ::= <optSign> "|" <instOperandBaseV> "|"

    <instOperandS>           ::= <instOperandAbsS>

    <instOperandAbsS>        ::= <optSign> "|" <instOperandBaseS> "|"


    <instOperandAddrVNS>     ::= <addrUseVNS>

    <instResult>             ::= <instResultCC>

    <instResultCC>           ::= <instResultBase> <ccMask>

    <instResultAddr>         ::= <instResultAddrCC>

    <instResultAddrCC>       ::= <instResultAddrBase> <ccMask>

    <branchLabel>            ::= <identifier>

    <paramUseV>              ::= <constantScalar>
                                   (*** instead of <constantScalar>
                                        <swizzleSuffix>)

    <paramUseS>              ::= <constantScalar>
                                   (*** instead of <constantScalar>
                                        <scalarSuffix>)

    <resultVtxBasic>         ::= "clip" "[" <clipPlaneNum> "]"

    <addrUseVNS>             ::= <addrVarName>

    <addrUseW>               ::= <addrVarName> <optAddrWriteMask>
                                   (*** instead of <addrVarName>
                                        <addrWriteMask>)

    <ccMask>                 ::= "(" <ccTest> ")"

    <ccTest>                 ::= <ccMaskRule> <swizzleSuffix>
```

```
<ccMaskRule>            ::= "EQ"
                          | "GE"
                          | "GT"
                          | "LE"
                          | "LT"
                          | "NE"
                          | "TR"
                          | "FL"

<optAddrWriteMask>      ::= <optWriteMask>
                              (*** instead of "." "x")

<addrComponent>         ::= <xyzwComponent>
                              (*** instead of "x")
```

(modify description of reserved identifiers)

... The following strings are reserved keywords and may not be used as identifiers:

```
    ABS, ADD, ADDRESS, ALIAS, ARA, ARL, ARR, ATTRIB, BRA, CAL, COS,
    DP3, DP4, DPH, DST, END, EX2, EXP, FLR, FRC, LG2, LIT, LOG, MAD,
    MAX, MIN, MOV, MUL, OPTION, OUTPUT, PARAM, POW, RCC, RCP, RET,
    RSQ, SEQ, SFL, SGE, SGT, SIN, SLE, SLT, SNE, SUB, SSG, STR, SWZ,
    TEMP, XPD, program, result, state, and vertex.
```

Add to Section 2.14.3.4, Vertex Program Results

(add to binding table)

Binding	Components	Description
result.clip[n]	(d,*,*,*)	clip plane distance

(add a paragraph before the last one) If a result variable binding matches "result.clip[n]", updates to the "x" component of the result variable set the clip distance for clip plane <n>.

(modify last paragraph) When in vertex program mode, all attributes of a transformed vertex, except for clip distances, are undefined at each vertex program invocation. Any results, or even individual components of results, that are not written to during vertex program execution remain undefined. All clip distances are initially zero, and remain zero if not written by the vertex program.

Modify Section 2.14.3.5, Vertex Program Address Registers

(modify first paragraph) Vertex program address register variables are a set of four-component signed integer vectors. Address registers are used as indices when performing relative addressing in program parameter arrays (section 2.14.4.2).

(modify third paragraph) Vertex program address register variables are undefined at each vertex program invocation. Address registers can be written by the ARA, ARL, and ARL instructions (section 2.14.5), and will be read by the ARA instruction and when a program uses relative addressing in program parameter arrays.

Add New Section 2.14.3.X, Condition Code Register (insert after Section 2.14.3.5, Vertex Program Address Registers)

The vertex program condition code register is a single four-component vector. Each component of this register is one of four enumerated values: GT (greater than), EQ (equal), LT (less than), or UN (unordered). The condition code register can be used to mask writes to registers and to evaluate conditional branches.

Most vertex program instructions can optionally update the condition code register. When a vertex program instruction updates the condition code register, a condition code component is set to LT if the corresponding component of the result is less than zero, EQ if it is equal to zero, GT if it is greater than zero, and UN if it is NaN (not a number).

The condition code register is initialized to a vector of EQ values each time a vertex program executes.

Modify Section 2.14.4, Vertex Program Execution Environment

(modify 3rd paragraph) Vertex programs execute a sequence of instructions, with support for conditional and unconditional branches, subroutine calls, and returns. Vertex programs begin by executing the instruction following the label "main". If no label "main" is defined, execution begins at the first instruction in the program. Instructions are executed in the order specified in the program, jumping when specified in branch instructions, until the end of the program is reached.

(modify instruction table) There are forty-two vertex program instructions. Vertex program instructions may have an optional suffix of "C" to allow an update of the condition code register (section 2.14.3.X). For example, there are two instructions to perform vector addition, "ADD" and "ADDC". The instructions and their respective input and output parameters are summarized in Table X.5.

Instruction	Inputs	Output	Description
ABS[C]	v	v	absolute value
ADD[C]	v,v	v	add
ARA[C]	a	a	address register add
ARL[C]	s	a	address register load
ARR[C]	v	a	address register load (round)
BRA	c	-	branch
CAL	c	-	subroutine call
COS[C]	s	ssss	cosine
DP3[C]	v,v	ssss	3-component dot product
DP4[C]	v,v	ssss	4-component dot product
DPH[C]	v,v	ssss	homogeneous dot product
DST[C]	v,v	v	distance vector
EX2[C]	s	ssss	exponential base 2
EXP[C]	s	v	exponential base 2 (approximate)
FLR[C]	v	v	floor
FRC[C]	v	v	fraction
LG2[C]	s	ssss	logarithm base 2
LIT[C]	v	v	compute light coefficients
LOG[C]	s	v	logarithm base 2 (approximate)
MAD[C]	v,v,v	v	multiply and add
MAX[C]	v,v	v	maximum
MIN[C]	v,v	v	minimum
MOV[C]	v	v	move
MUL[C]	v,v	v	multiply
POW[C]	s,s	ssss	exponentiate
RCC[C]	s	ssss	reciprocal (clamped)
RCP[C]	s	ssss	reciprocal
RET	c	-	subroutine return
RSQ[C]	s	ssss	reciprocal square root
SEQ[C]	v,v	v	set on equal
SFL[C]	v,v	v	set on false
SGE[C]	v,v	v	set on greater than or equal
SGT[C]	v,v	v	set on greater than
SIN[C]	s	ssss	sine
SLE[C]	v,v	v	set on less than or equal
SLT[C]	v,v	v	set on less than
SNE[C]	v,v	v	set on not equal
SSG[C]	v	v	set sign
STR[C]	v,v	v	set on true
SUB[C]	v,v	v	subtract
SWZ[C]	v	v	extended swizzle
XPD[C]	v,v	v	cross product

Table X.5: Summary of vertex program instructions. "[C]" indicates that the opcode supports the condition code update modifier. "v" indicates a floating-point vector input or output, "s" indicates a floating-point scalar input, "ssss" indicates a scalar output replicated across a 4-component result vector, "a" indicates a vector address register, and "c" indicates a condition code test.

Modify Section 2.14.4.1, Vertex Program Operands

(add prior to the discussion of negation) A component-wise absolute value operation can optionally performed on the operand if the operand is surrounded with two "|" characters. For example, "|src|" indicates

```
    that a component-wise absolute value operation should be performed on
    the variable named "src".  In terms of the grammar, this operation
    is performed if the <instOperandV> or <instOperandS> grammar rules
    match <instOperandAbsV> or <instOperandAbsS>, respectively.

    (modify operand load pseudo-code) The following pseudo-code spells
    out the operand generation process.  In the example, "float" is a
    floating-point scalar type, while "floatVec" is a four-component
    vector.  "source" refers to the register used for the operand,
    matching the <srcReg> rule.  "abs" is TRUE if an absolute value
    operation should be performed on the operand (<instOperandAbsV> or
    <instOperandAbsS> rules) "negate" is TRUE if the <optionalSign> rule
    in <scalarSrcReg> or <swizzleSrcReg> matches "-" and FALSE otherwise.
    The ".c***", ".*c**", ".**c*", ".***c" modifiers refer to the x,
    y, z, and w components obtained by the swizzle operation; the ".c"
    modifier refers to the single component selected for a scalar load.

      floatVec VectorLoad(floatVec source)
      {
          floatVec operand;

          operand.x = source.c***;
          operand.y = source.*c**;
          operand.z = source.**c*;
          operand.w = source.***c;
          if (abs) {
             operand.x = abs(operand.x);
             operand.y = abs(operand.y);
             operand.z = abs(operand.z);
             operand.w = abs(operand.w);
          }
          if (negate) {
             operand.x = -operand.x;
             operand.y = -operand.y;
             operand.z = -operand.z;
             operand.w = -operand.w;
          }

          return operand;
      }

      float ScalarLoad(floatVec source)
      {
          float operand;

          operand = source.c;
          if (abs) {
            operand = abs(operand);
          if (negate) {
            operand = -operand;
          }

          return operand;
      }
```

Rewrite Section 2.14.4.3, Vertex Program Destination Register Update

Most vertex program instructions write a 4-component result vector to a single temporary or vertex result register. Writes to individual components of the destination register are controlled by individual component write masks specified as part of the instruction.

The component write mask is specified by the <optionalMask> rule found in the <maskedDstReg> rule. If the optional mask is "", all components are enabled. Otherwise, the optional mask names the individual components to enable. The characters "x", "y", "z", and "w" match the x, y, z, and w components respectively. For example, an optional mask of ".xzw" indicates that the x, z, and w components should be enabled for writing but the y component should not. The grammar requires that the destination register mask components must be listed in "xyzw" order.

The condition code write mask is specified by the <ccMask> rule found in the <instResultCC> and <instResultAddrCC> rules. The condition code register is loaded and swizzled according to the swizzle codes specified by <swizzleSuffix>. Each component of the swizzled condition code is tested according to the rule given by <ccMaskRule>. <ccMaskRule> may have the values "EQ", "NE", "LT", "GE", LE", or "GT", which mean to enable writes if the corresponding condition code field evaluates to equal, not equal, less than, greater than or equal, less than or equal, or greater than, respectively. Comparisons involving condition codes of "UN" (unordered) evaluate to true for "NE" and false otherwise. For example, if the condition code is (GT,LT,EQ,GT) and the condition code mask is "(NE.zyxw)", the swizzle operation will load (EQ,LT,GT,GT) and the mask will thus will enable writes on the y, z, and w components. In addition, "TR" always enables writes and "FL" always disables writes, regardless of the condition code. If the condition code mask is empty, it is treated as "(TR)".

Each component of the destination register is updated with the result of the vertex program instruction if and only if the component is enabled for writes by both the component write mask and the condition code write mask. Otherwise, the component of the destination register remains unchanged.

A vertex program instruction can also optionally update the condition code register. The condition code is updated if the condition code register update suffix "C" is present in the instruction. The instruction "ADDC" will update the condition code; the otherwise equivalent instruction "ADD" will not. If condition code updates are enabled, each component of the destination register enabled for writes is compared to zero. The corresponding component of the condition code is set to "LT", "EQ", or "GT", if the written component is less than, equal to, or greater than zero, respectively. Condition code components are set to "UN" if the written component is NaN (not a number). Values of -0.0 and +0.0 both evaluate to "EQ". If a component of the destination register is not enabled for writes, the corresponding condition code component is also unchanged.

In the following example code,

```
    # R1=(-2, 0, 2, NaN)              R0                  CC
    MOVC R0, R1;               # ( -2,  0,   2, NaN) (LT,EQ,GT,UN)
    MOVC R0.xyz, R1.yzwx;      # (  0,  2, NaN, NaN) (EQ,GT,UN,UN)
    MOVC R0 (NE), R1.zywx;     # (  0,  0, NaN,  -2) (EQ,EQ,UN,LT)
```

the first instruction writes (-2,0,2,NaN) to R0 and updates the condition code to (LT,EQ,GT,UN). The second instruction, only the "x", "y", and "z" components of R0 and the condition code are updated, so R0 ends up with (0,2,NaN,NaN) and the condition code ends up with (EQ,GT,UN,UN). In the third instruction, the condition code mask disables writes to the x component (its condition code field is "EQ"), so R0 ends up with (0,0,NaN,-2) and the condition code ends up with (EQ,EQ,UN,LT).

The following pseudocode illustrates the process of writing a result vector to the destination register. In the pseudocode, "instrmask" refers to the component write mask given by the <optWriteMask> rule. "ccMaskRule" refers to the condition code mask rule given by <ccMask> and "updatecc" is TRUE if and only if condition code updates are enabled. "result", "destination", and "cc" refer to the result vector, the register selected by <dstRegister> and the condition code, respectively. Condition codes do not exist in the VP1 execution environment.

```
  boolean TestCC(CondCode field) {
      switch (ccMaskRule) {
      case "EQ":  return (field == "EQ");
      case "NE":  return (field != "EQ");
      case "LT":  return (field == "LT");
      case "GE":  return (field == "GT" || field == "EQ");
      case "LE":  return (field == "LT" || field == "EQ");
      case "GT":  return (field == "GT");
      case "TR":  return TRUE;
      case "FL":  return FALSE;
      case "":    return TRUE;
      }
  }

  enum GenerateCC(float value) {
    if (value == NaN) {
      return UN;
    } else if (value < 0) {
      return LT;
    } else if (value == 0) {
      return EQ;
    } else {
      return GT;
    }
  }
  void UpdateDestination(floatVec destination, floatVec result)
  {
      floatVec merged;
      ccVec    mergedCC;

      // Merge the converted result into the destination register, under
      // control of the compile- and run-time write masks.
      merged = destination;
      mergedCC = cc;
      if (instrMask.x && TestCC(cc.c***)) {
          merged.x = result.x;
          if (updatecc) mergedCC.x = GenerateCC(result.x);
      }
      if (instrMask.y && TestCC(cc.*c**)) {
          merged.y = result.y;
          if (updatecc) mergedCC.y = GenerateCC(result.y);
      }
      if (instrMask.z && TestCC(cc.**c*)) {
          merged.z = result.z;
          if (updatecc) mergedCC.z = GenerateCC(result.z);
      }
      if (instrMask.w && TestCC(cc.***c)) {
          merged.w = result.w;
          if (updatecc) mergedCC.w = GenerateCC(result.w);
      }

      // Write out the new destination register and condition code.
      destination = merged;
      cc = mergedCC;
  }
```

While this rule describes floating-point results, the same logic applies to the integer results generated by the ARA, ARL, and ARR instructions.

Add Section 2.14.4.X, Vertex Program Branching (before Section 2.14.4.4, Vertex Program Result Processing)

Vertex programs can contain one or more instruction labels, matching the grammar rule <branchLabel>. An instruction label can be referred to explicitly in branch (BRA) or subroutine call (CAL) instructions. Instruction labels can be defined or used at any point in the body of a program, and can be used in instructions before being defined in the program string.

Branching instructions can be conditional. The branch condition is specified by the <optBranchCond> grammar rule and may depend on the contents of the condition code register. Branch conditions are evaluated by evaluating a condition code write mask in exactly the same manner as done for register writes (section 2.14.2.2). If any of the four components of the condition code write mask are enabled, the branch is taken and execution continues with the instruction following the label specified in the instruction. Otherwise, the instruction is ignored and vertex program execution continues with the next instruction. In the following example code,

```
      MOVC CC, c[0];         # c[0]=(-2, 0, 2, NaN), CC gets (LT,EQ,GT,UN)
      BRA label1 (LT.xyzw);
      MOV R0,R1;             # not executed
    label1:
      BRA label2 (LT.wyzw);
      MOV R0,R2;             # executed
    label2:
```

the first BRA instruction loads a condition code of (LT,EQ,GT,UN) while the second BRA instruction loads a condition code of (UN,EQ,GT,UN). The first branch will be taken because the "x" component evaluates to LT; the second branch will not be taken because no component evaluates to LT.

Vertex programs can specify subroutine calls. When a subroutine call (CAL) instruction is executed, a reference to the instruction immediately following the CAL instruction is pushed onto the call stack. When a subroutine return (RET) instruction is executed, an instruction reference is popped off the call stack and program execution continues with the popped instruction. A vertex program will terminate if a CAL instruction is executed with MAX_PROGRAM_CALL_DEPTH_NV entries already in the call stack or if a RET instruction is executed with an empty call stack.

If a vertex program has an instruction label "main", program execution begins with the instruction immediately following the instruction label. Otherwise, program execution begins with the first instruction of the program. Instructions will be executed sequentially in the order specified in the program, although branch instructions will affect the instruction execution order, as described above. A vertex program will terminate after executing a RET instruction with an empty call stack. A vertex program will also terminate after executing the last instruction in the program, unless that instruction was a taken branch.

A vertex program will fail to load if an instruction refers to a label that is not defined in the program string.

A vertex program will terminate abnormally if a subroutine call instruction produces a call stack overflow. Additionally, a vertex program will terminate abnormally after executing MAX_PROGRAM_EXEC_INSTRUCTIONS instructions to prevent hangs caused by infinite loops in the program.

When a vertex program terminates, normally or abnormally, it will emit a vertex whose attributes are taken from the final values of the vertex result registers (section 2.14.1.5).

Modify Section 2.14.4.4, Vertex Program Result Processing

(modify 3rd paragraph) Transformed vertices are then assembled into primitives and clipped as described in section 2.11. Clip distance results are used to control user clip planes.

Add to Section 2.14.4.5, Vertex Program Options:

Section 2.14.4.5.2, NV_vertex_program2 Option

If a vertex program specifies the "NV_vertex_program2" program option, the grammar will be extended to support the features found in the NV_vertex_program2 extension not present in the ARB_vertex_program extension, including:

* the availability of the following instructions:
    - ARA (address register add, useful for looping),
    - ARR (address register load with round),
    - BRA (branch),
    - CAL (subroutine call),
    - COS (cosine),
    - RET (subroutine return),
    - SEQ (set on equal),
    - SFL (set on false),
    - SGT (set on greater than),
    - SIN (sine),
    - SLE (set on less than or equal),
    - SNE (set on not equal),
    - SSG (set sign), and
    - STR (set on true).
* up to MAX_CALL_DEPTH_NV levels of subroutine calls/returns,
* a four-component condition code register to hold the sign of result vector components (useful for comparisons),
* a condition code update opcode suffix "C", where the results of the instruction are used to update the condition code register,
* a condition code write mask operator, where the condition code register is swizzled and tested, and the test results are used to mask register writes,
* six clip distance result bindings that can be used to perform more complicated user clipping operations than those provided with the position invariant program option,
* four-component address registers (instead of one-component registers in ARB_vertex_program), with the "ARL" instruction extended to produce a vector result,
* an absolute value operator on scalar and swizzled operands.

The added functionality is identical to that provided by NV_vertex_program2 extension specification.

Modify Section 2.14.5.3, ARL: Address Register Load

The ARL instruction loads a single vector operand and performs a component-wise floor operation to generate a signed integer result vector.

```
  tmp = VectorLoad(op0);
  iresult.x = floor(tmp.x);
  iresult.y = floor(tmp.y);
  iresult.z = floor(tmp.z);
  iresult.w = floor(tmp.w);
```

The floor operation returns the largest integer less than or equal to the operand. For example floor(-1.7) = -2.0, floor(+1.0) = +1.0, and floor(+3.7) = +3.0.

Note that in the unextended ARB_vertex_program specification, the ARL instruction loads a scalar operand and generates a scalar result.

Add to Section 2.14.5, Vertex Program Instruction Set

Section 2.14.5.28, ARA: Address Register Add

The ARA instruction adds two pairs of components of a vector address register operand to produce an integer result vector. The "x" and "z" components of the result vector contain the sum of the "x" and "z" components of the operand; the "y" and "w" components of the result vector contain the sum of the "y" and "w" components of the operand.

```
  itmp = AddrVectorLoad(op0);
  iresult.x = itmp.x + itmp.z;
  iresult.y = itmp.y + itmp.w;
  iresult.z = itmp.x + itmp.z;
  iresult.w = itmp.y + itmp.w;
```

Component swizzling is not supported when the operand is loaded.

Section 2.14.5.29, ARR: Address Register Load (with round)

The ARR instruction loads a single vector operand and performs a component-wise round operation to generate a signed integer result vector.

```
  tmp = VectorLoad(op0);
  iresult.x = round(tmp.x);
  iresult.y = round(tmp.y);
  iresult.z = round(tmp.z);
  iresult.w = round(tmp.w);
```

The round operation returns the nearest integer to the operand. If the fractional portion of the operand is 0.5, round() selects the nearest even integer. For example round(-1.7) = -2.0, round(+1.0) = +1.0, and round(+3.7) = +4.0.

Section 2.14.5.30, BRA: Branch

The BRA instruction conditionally transfers control to the instruction following the label specified in the instruction. The following pseudocode describes the operation of the instruction:

```
  if (TestCC(cc.c***) || TestCC(cc.*c**) ||
      TestCC(cc.**c*) || TestCC(cc.***c)) {
    // continue execution at instruction following <branchLabel>
  } else {
    // do nothing
  }
```

In the pseudocode, <branchLabel> is the label specified in the instruction according to the <branchLabel> grammar rule.

Section 2.14.5.31, CAL: Subroutine Call

The CAL instruction conditionally transfers control to the instruction following the label specified in the instruction. It also pushes a reference to the instruction immediately following the CAL instruction onto the call stack, where execution will continue after executing the matching RET instruction. The following pseudocode describes the operation of the instruction:

```
  if (TestCC(cc.c***) || TestCC(cc.*c**) ||
      TestCC(cc.**c*) || TestCC(cc.***c)) {
    if (callStackDepth >= MAX_PROGRAM_CALL_DEPTH_NV) {
      // terminate vertex program
    } else {
      callStack[callStackDepth] = nextInstruction;
      callStackDepth++;
    }
    // continue execution at instruction following <branchLabel>
  } else {
    // do nothing
  }
```

In the pseudocode, <branchLabel> is the label specified in the instruction matching the <branchLabel> grammar rule, <callStackDepth> is the current depth of the call stack, <callStack> is an array holding the call stack, and <nextInstruction> is a reference to the instruction immediately following the present one in the program string.

If the call stack overflows, the vertex program terminates abnormally and all vertex program results are undefined.

Section 2.14.5.32, COS: Cosine

The COS instruction approximates the cosine of the angle specified by the scalar operand and replicates the approximation to all four components of the result vector. The angle is specified in radians and does not have to be in the range [0,2*PI].

```
  tmp = ScalarLoad(op0);
  result.x = ApproxCosine(tmp);
  result.y = ApproxCosine(tmp);
  result.z = ApproxCosine(tmp);
  result.w = ApproxCosine(tmp);
```

Section 2.14.5.33, RCC: Reciprocal (Clamped)

The RCC instruction approximates the reciprocal of the scalar operand, clamps the result to one of two ranges, and replicates the clamped result to all four components of the result vector.

If the approximated reciprocal is greater than 0.0, the result is clamped to the range [2^-64, 2^+64]. If the approximate reciprocal is not greater than zero, the result is clamped to the range [-2^+64, -2^-64].

```
  tmp = ScalarLoad(op0);
  result.x = ClampApproxReciprocal(tmp);
  result.y = ClampApproxReciprocal(tmp);
  result.z = ClampApproxReciprocal(tmp);
  result.w = ClampApproxReciprocal(tmp);
```

The following rule applies to reciprocation:

  1. ApproxReciprocal(+1.0) = +1.0.

Section 2.14.5.34, RET: Subroutine Call Return

The RET instruction conditionally returns from a subroutine initiated by a CAL instruction by popping an instruction reference off the top of the call stack and transferring control to the referenced instruction. The following pseudocode describes the operation of the instruction:

```
  if (TestCC(cc.c***) || TestCC(cc.*c**) ||
      TestCC(cc.**c*) || TestCC(cc.***c)) {
    if (callStackDepth <= 0) {
      // terminate vertex program
    } else {
      callStackDepth--;
      instruction = callStack[callStackDepth];
    }

    // continue execution at <instruction>
  } else {
    // do nothing
  }
```

In the pseudocode, <callStackDepth> is the depth of the call stack, <callStack> is an array holding the call stack, and <instruction> is a reference to an instruction previously pushed onto the call stack.

If the call stack is empty when RET executes, the vertex program terminates normally.

Section 2.14.5.35, SEQ: Set on Equal

The SEQ instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operand is equal to that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x == tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y == tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z == tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w == tmp1.w) ? 1.0 : 0.0;
```

Section 2.14.5.36, SFL: Set on False

The SFL instruction is a degenerate case of the other "Set on" instructions that sets all components of the result vector to 0.0.

```
  result.x = 0.0;
  result.y = 0.0;
  result.z = 0.0;
  result.w = 0.0;
```

Section 2.14.5.37, SGT: Set on Greater Than

The SGT instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operands is greater than that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x > tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y > tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z > tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w > tmp1.w) ? 1.0 : 0.0;
```

Section 2.14.5.38, SIN: Sine

The SIN instruction approximates the sine of the angle specified by the scalar operand and replicates it to all four components of the result vector. The angle is specified in radians and does not have to be in the range [0,2*PI].

```
  tmp = ScalarLoad(op0);
  result.x = ApproxSine(tmp);
  result.y = ApproxSine(tmp);
  result.z = ApproxSine(tmp);
  result.w = ApproxSine(tmp);
```

Section 2.14.5.39, SLE: Set on Less Than or Equal

The SLE instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operand is less than or equal to that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x <= tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y <= tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z <= tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w <= tmp1.w) ? 1.0 : 0.0;
```

Section 2.14.5.40, SNE: Set on Not Equal

The SNE instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operand is not equal to that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x != tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y != tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z != tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w != tmp1.w) ? 1.0 : 0.0;
```

Section 2.14.5.41, SSG: Set Sign

The SSG instruction generates a result vector containing the signs of each component of the single vector operand. Each component of the result vector is 1.0 if the corresponding component of the operand is greater than zero, 0.0 if the corresponding component of the operand is equal to zero, and -1.0 if the corresponding component of the operand is less than zero.

```
  tmp = VectorLoad(op0);
  result.x = SetSign(tmp.x);
  result.y = SetSign(tmp.y);
  result.z = SetSign(tmp.z);
  result.w = SetSign(tmp.w);
```

Section 2.14.5.42, STR: Set on True

The STR instruction is a degenerate case of the other "Set on" instructions that sets all components of the result vector to 1.0.

```
  result.x = 1.0;
  result.y = 1.0;
  result.z = 1.0;
  result.w = 1.0;
```

Additions to Chapter 3 of the OpenGL 1.4 Specification (Rasterization)

None.

Additions to Chapter 4 of the OpenGL 1.4 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 1.4 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 1.4 Specification (State and State Requests)

None.

Additions to Appendix A of the OpenGL 1.4 Specification (Invariance)

None.

Additions to the AGL/GLX/WGL Specifications

None.

Dependencies on ARB_vertex_program

This specification is based on a modified version of the grammar published in the ARB_vertex_program specification. This modified grammar (see below) includes a few structural changes to better accommodate new functionality from this and other extensions, but should be functionally equivalent to the ARB_vertex_program grammar.

```
    <program>              ::= <optionSequence> <statementSequence> "END"

    <optionSequence>       ::= <optionSequence> <option>
                             | /* empty */

    <option>               ::= "OPTION" <optionName> ";"

    <optionName>           ::= "ARB_position_invariant"

    <statementSequence>    ::= <statement> <statementSequence>
                             | /* empty */

    <statement>            ::= <instruction> ";"
                             | <namingStatement> ";"

    <instruction>          ::= <ALUInstruction>

    <ALUInstruction>       ::= <VECTORop_instruction>
                             | <SCALARop_instruction>
                             | <BINSCop_instruction>
                             | <BINop_instruction>
                             | <TRIop_instruction>
                             | <SWZop_instruction>
                             | <ARLop_instruction>

    <VECTORop_instruction>  ::= <VECTORop> <instResult> "," <instOperandV>

    <VECTORop>             ::= "ABS"
                             | "FLR"
                             | "FRC"
                             | "LIT"
                             | "MOV"

    <SCALARop_instruction>  ::= <SCALARop> <instResult> "," <instOperandS>
    <SCALARop>                ::= "EX2"
                                | "EXP"
                                | "LG2"
                                | "LOG"
                                | "RCP"
                                | "RSQ"

    <BINSCop_instruction>     ::= <BINSCop> <instResult> "," <instOperandS> ","
                                  <instOperandS>

    <BINSCop>                 ::= "POW"

    <BINop_instruction>       ::= <BINop> <instResult> "," <instOperandV> ","
                                  <instOperandV>

    <BINop>                   ::= "ADD"
                                | "DP3"
                                | "DP4"
                                | "DPH"
                                | "DST"
                                | "MAX"
                                | "MIN"
                                | "MUL"
                                | "SGE"
                                | "SLT"
                                | "SUB"
                                | "XPD"

    <TRIop_instruction>       ::= <TRIop> <instResult> "," <instOperandV> ","
                                  <instOperandV> "," <instOperandV>

    <TRIop>                   ::= "MAD"

    <SWZop_instruction>       ::= <SWZop> <instResult> "," <instOperandVNS> ","
                                  <extendedSwizzle>

    <SWZop>                   ::= "SWZ"

    <ARLop_instruction>       ::= <ARLop> <instResultAddr> "," <ARLop_src>

    <ARLop>                   ::= "ARL"

    <ARLop_src>               ::= <instOperandS>

    <instOperandV>            ::= <instOperandBaseV>

    <instOperandBaseV>        ::= <optSign> <attribUseV>
                                | <optSign> <tempUseV>
                                | <optSign> <paramUseV>

    <instOperandS>            ::= <instOperandBaseS>

    <instOperandBaseS>        ::= <optSign> <attribUseS>
                                | <optSign> <tempUseS>
                                | <optSign> <paramUseS>
    <instOperandVNS>        ::= <attribVNS>
                              | <tempUseVNS>
                              | <paramUseVNS>

    <instResult>            ::= <instResultBase>

    <instResultBase>        ::= <tempUseW>
                              | <resultUseW>

    <instResultAddr>        ::= <instResultAddrBase>

    <instResultAddrBase>    ::= <addrUseW>

    <namingStatement>       ::= <ATTRIB_statement>
                              | <PARAM_statement>
                              | <TEMP_statement>
                              | <OUTPUT_statement>
                              | <ALIAS_statement>
                              | <ADDRESS_statement>

    <ATTRIB_statement>      ::= "ATTRIB" <establishName> "=" <attribUseD>

    <PARAM_statement>       ::= <PARAM_singleStmt>
                              | <PARAM_multipleStmt>

    <PARAM_singleStmt>      ::= "PARAM" <establishName> <paramSingleInit>

    <PARAM_multipleStmt>    ::= "PARAM" <establishName> "[" <optArraySize> "]"
                                <paramMultipleInit>

    <optArraySize>          ::= /* empty */
                              | <integer> /* [1,MAX_PROGRAM_PARAMETERS_ARB]*/

    <paramSingleInit>       ::= "=" <paramUseDB>

    <paramMultipleInit>     ::= "=" "{" <paramMultInitList> "}"

    <paramMultInitList>     ::= <paramUseDM>
                              | <paramUseDM> "," <paramMultInitList>

    <TEMP_statement>        ::= "TEMP" <varNameList>

    <OUTPUT_statement>      ::= "OUTPUT" <establishName> "=" <resultUseD>

    <ALIAS_statement>       ::= "ALIAS" <establishName> "=" <establishedName>

    <establishedName>       ::= <tempVarName>
                              | <addrVarName>
                              | <attribVarName>
                              | <paramArrayVarName>
                              | <paramSingleVarName>
                              | <resultVarName>

    <ADDRESS_statement>     ::= "ADDRESS" <varNameList>

    <varNameList>           ::= <establishName>
                              | <establishName> "," <varNameList>
    <establishName>         ::= <identifier>

    <attribUseV>            ::= <attribBasic> <swizzleSuffix>
                              | <attribVarName> <swizzleSuffix>
                              | <attribColor> <swizzleSuffix>
                              | <attribColor> "." <colorType> <swizzleSuffix>

    <attribUseS>            ::= <attribBasic> <scalarSuffix>
                              | <attribVarName> <scalarSuffix>
                              | <attribColor> <scalarSuffix>
                              | <attribColor> "." <colorType> <scalarSuffix>

    <attribUseVNS>          ::= <attribBasic>
                              | <attribVarName>
                              | <attribColor>
                              | <attribColor> "." <colorType>

    <attribUseD>            ::= <attribBasic>
                              | <attribColor>
                              | <attribColor> "." <colorType>

    <attribBasic>           ::= "vertex" "." <attribVtxBasic>

    <attribVtxBasic>        ::= "position"
                              | "weight" <vtxOptWeightNum>
                              | "normal"
                              | "fogcoord"
                              | "texcoord" <optTexCoordNum>
                              | "matrixindex" "[" <vtxWeightNum> "]"
                              | "attrib" "[" <vtxAttribNum> "]"

    <attribColor>           ::= "vertex" "." "color"

    <paramUseV>             ::= <paramSingleVarName> <swizzleSuffix>
                              | <paramArrayVarName> "[" <arrayMem> "]"
                                <swizzleSuffix>
                              | <stateSingleItem> <swizzleSuffix>
                              | <programSingleItem> <swizzleSuffix>
                              | <constantVector> <swizzleSuffix>
                              | <constantScalar> <swizzleSuffix>

    <paramUseS>             ::= <paramSingleVarName> <scalarSuffix>
                              | <paramArrayVarName> "[" <arrayMem> "]"
                                <scalarSuffix>
                              | <stateSingleItem> <scalarSuffix>
                              | <programSingleItem> <scalarSuffix>
                              | <constantVector> <scalarSuffix>
                              | <constantScalar> <scalarSuffix>

    <paramUseVNS>           ::= <paramSingleVarName>
                              | <paramArrayVarName> "[" <arrayMem> "]"
                              | <stateSingleItem>
                              | <programSingleItem>
                              | <constantVector>
                              | <constantScalar>
    <paramUseDB>            ::= <stateSingleItem>
                              | <programSingleItem>
                              | <constantVector>
                              | <signedConstantScalar>

    <paramUseDM>            ::= <stateMultipleItem>
                              | <programMultipleItem>
                              | <constantVector>
                              | <signedConstantScalar>

    <stateMultipleItem>     ::= <stateSingleItem>
                              | "state" "." <stateMatrixRows>

    <stateSingleItem>       ::= "state" "." <stateMaterialItem>
                              | "state" "." <stateLightItem>
                              | "state" "." <stateLightModelItem>
                              | "state" "." <stateLightProdItem>
                              | "state" "." <stateFogItem>
                              | "state" "." <stateMatrixRow>
                              | "state" "." <stateTexGenItem>
                              | "state" "." <stateClipPlaneItem>
                              | "state" "." <statePointItem>

    <stateMaterialItem>     ::= "material" "." <stateMatProperty>
                              | "material" "." <faceType> "."
                                <stateMatProperty>

    <stateMatProperty>      ::= "ambient"
                              | "diffuse"
                              | "specular"
                              | "emission"
                              | "shininess"

    <stateLightItem>        ::= "light" "[" <stateLightNumber> "]" "."
                                <stateLightProperty>

    <stateLightProperty>    ::= "ambient"
                              | "diffuse"
                              | "specular"
                              | "position"
                              | "attenuation"
                              | "spot" "." <stateSpotProperty>
                              | "half"

    <stateSpotProperty>     ::= "direction"

    <stateLightModelItem>   ::= "lightmodel" <stateLModProperty>

    <stateLModProperty>     ::= "." "ambient"
                              | "." "scenecolor"
                              | "." <faceType> "." "scenecolor"

    <stateLightProdItem>    ::= "lightprod" "[" <stateLightNumber> "]" "."
                                <stateLProdProperty>
                              | "lightprod" "[" <stateLightNumber> "]" "."
                                <faceType> "." <stateLProdProperty>
    <stateLProdProperty>    ::= "ambient"
                              | "diffuse"
                              | "specular"

    <stateLightNumber>      ::= <integer> /* [0,MAX_LIGHTS-1] */

    <stateFogItem>          ::= "fog" "." <stateFogProperty>

    <stateFogProperty>      ::= "color"
                              | "params"

    <stateMatrixRows>       ::= <stateMatrixItem>
                              | <stateMatrixItem> "." <stateMatModifier>
                              | <stateMatrixItem> "." "row" "["
                                <stateMatrixRowNum> ".." <stateMatrixRowNum>
                                "]"
                              | <stateMatrixItem> "." <stateMatModifier> "."
                                "row" "[" <stateMatrixRowNum> ".."
                                <stateMatrixRowNum> "]"

    <stateMatrixRow>        ::= <stateMatrixItem> "." "row" "["
                                <stateMatrixRowNum> "]"
                              | <stateMatrixItem> "." <stateMatModifier> "."
                                "row" "[" <stateMatrixRowNum> "]"

    <stateMatrixItem>       ::= "matrix" "." <stateMatrixName>

    <stateMatModifier>      ::= "inverse"
                              | "transpose"
                              | "invtrans"

    <stateMatrixName>       ::= "modelview" <stateOptModMatNum>
                              | "projection"
                              | "mvp"
                              | "texture" <optTexCoordNum>
                              | "palette" "[" <statePaletteMatNum> "]"
                              | "program" "[" <stateProgramMatNum> "]"

    <stateMatrixRowNum>     ::= <integer> /* [0,3] */

    <stateOptModMatNum>     ::= /* empty */
                              | "[" <stateModMatNum> "]"

    <stateModMatNum>        ::= <integer> /*[0,MAX_VERTEX_UNITS_ARB-1]*/

    <statePaletteMatNum>    ::= <integer> /*[0,MAX_PALETTE_MATRICES_ARB-1]*/

    <stateProgramMatNum>    ::= <integer> /*[0,MAX_PROGRAM_MATRICES_ARB-1]*/

    <stateTexGenItem>       ::= "texgen" <optTexCoordNum> "."
                                <stateTexGenType> "." <stateTexGenCoord>

    <stateTexGenType>       ::= "eye"
                              | "object"
    <stateTexGenCoord>      ::= "s"
                              | "t"
                              | "r"
                              | "q"

    <stateClipPlaneItem>    ::= "clip" "[" <clipPlaneNum> "]" "." "plane"

    <statePointItem>        ::= "point" "." <statePointProperty>

    <statePointProperty>    ::= "size"
                              | "attenuation"

    <programSingleItem>     ::= <progEnvParam>
                              | <progLocalParam>

    <programMultipleItem>   ::= <progEnvParams>
                              | <progLocalParams>

    <progEnvParams>         ::= "program" "." "env" "[" <progEnvParamNums> "]"

    <progEnvParamNums>      ::= <progEnvParamNum>
                              | <progEnvParamNum> ".." <progEnvParamNum>

    <progEnvParam>          ::= "program" "." "env" "[" <progEnvParamNum> "]"

    <progLocalParams>       ::= "program" "." "local" "[" <progLocalParamNums>
                                "]"

    <progLocalParamNums>    ::= <progLocalParamNum>
                              | <progLocalParamNum> ".." <progLocalParamNum>

    <progLocalParam>        ::= "program" "." "local" "[" <progLocalParamNum>
                                "]"

    <progEnvParamNum>       ::= <integer>
                                /*[0,MAX_PROGRAM_ENV_PARAMETERS_ARB-1]*/

    <progLocalParamNum>     ::= <integer>
                                /*[0,MAX_PROGRAM_LOCAL_PARAMETERS_ARB-1]*/

    <constantVector>        ::= "{" <constantVectorList> "}"

    <constantVectorList>    ::= <signedConstantScalar>
                              | <signedConstantScalar> ","
                                <signedConstantScalar>
                              | <signedConstantScalar> ","
                                <signedConstantScalar> ","
                                <signedConstantScalar>
                              | <signedConstantScalar> ","
                                <signedConstantScalar> ","
                                <signedConstantScalar> ","
                                <signedConstantScalar>

    <signedConstantScalar>  ::= <optSign> <constantScalar>

    <constantScalar>        ::= <floatConstant>
    <floatConstant>         ::= <float>

    <tempUseV>              ::= <tempVarName> <swizzleSuffix>

    <tempUseS>              ::= <tempVarName> <scalarSuffix>

    <tempUseVNS>            ::= <tempVarName>

    <tempUseW>              ::= <tempVarName> <optWriteMask>

    <resultUseW>            ::= <resultBasic> <optWriteMask>
                              | <resultVarName> <optWriteMask>
                              | <resultVtxColor> <optWriteMask>
                              | <resultVtxColor> "." <colorType>
                                <optWriteMask>
                              | <resultVtxColor> "." <faceType> <optWriteMask>
                              | <resultVtxColor> "." <faceType> "."
                                <colorType> "." <optWriteMask>

    <resultUseD>            ::= <resultBasic>
                              | <resultVtxColor>
                              | <resultVtxColor> "." <colorType>
                              | <resultVtxColor> "." <faceType>
                              | <resultVtxColor> "." <faceType> "."
                                <colorType>

    <resultBasic>           ::= "result" "." <resultVtxBasic>

    <resultVtxBasic>        ::= "position"
                              | "fogcoord"
                              | "pointsize"
                              | "texcoord" <optTexCoordNum>

    <resultVtxColor>        ::= "result" "." "color"

    <arrayMem>              ::= <arrayMemAbs>
                              | <arrayMemRel>

    <arrayMemRel>           ::= <addrUseS> <arrayMemRelOffset>

    <arrayMemAbs>           ::= <integer>

    <arrayMemRelOffset>     ::= /* empty */
                              | "+" <integer>
                              | "-" <integer>

    <addrUseS>              ::= <addrVarName> <scalarAddrSuffix>

    <addrUseW>              ::= <addrVarName> <addrWriteMask>

    <addrWriteMask>         ::= "." "x"

    <optWriteMask>          ::= /* empty */
                              | <xyzwMask>
    <xyzwMask>              ::= "." "x"
                              | "." "y"
                              | "." "xy"
                              | "." "z"
                              | "." "xz"
                              | "." "yz"
                              | "." "xyz"
                              | "." "w"
                              | "." "xw"
                              | "." "yw"
                              | "." "xyw"
                              | "." "zw"
                              | "." "xzw"
                              | "." "yzw"
                              | "." "xyzw"

    <swizzleSuffix>         ::= /* empty */
                              | "." <component>
                              | "." <xyzwComponent> <xyzwComponent>
                                <xyzwComponent> <xyzwComponent>

    <extendedSwizzle>       ::= <extSwizComp> "," <extSwizComp> ","
                                <extSwizComp> "," <extSwizComp>

    <extSwizComp>           ::= <optSign> <xyzwExtSwizSel>

    <xyzwExtSwizSel>        ::= "0"
                              | "1"
                              | <xyzwComponent>

    <scalarAddrSuffix>      ::= "." <addrComponent>

    <addrComponent>         ::= "x"

    <scalarSuffix>          ::= "." <component>

    <component>             ::= <xyzwComponent>

    <xyzwComponent>         ::= "x"
                              | "y"
                              | "z"
                              | "w"

    <optSign>               ::= /* empty */
                              | "-"
                              | "+"

    <faceType>              ::= "front"
                              | "back"

    <colorType>             ::= "primary"
                              | "secondary"

    <vtxAttribNum>          ::= <integer> /*[0,MAX_VERTEX_ATTRIBS_ARB-1]*/

    <vtxOptWeightNum>       ::= /* empty */
                              | "[" <vtxWeightNum> "]"
    <vtxWeightNum>          ::= <integer> /*[0,MAX_VERTEX_UNITS_ARB-1] must be
                                divisible by four */

    <optTexCoordNum>        ::= /* empty */
                              | "[" <texCoordNum> "]"

    <texCoordNum>           ::= <integer> /*[0,MAX_TEXTURE_COORDS_ARB-1]*/

    <clipPlaneNum>          ::= <integer> /*[0,MAX_CLIP_PLANES-1]*/
```

The <integer>, <float>, and <identifier> grammar rules match integer constants, floating point constants, and identifier names as described in the ARB_vertex_program specification. The <float> grammar rule here is identical to the <floatConstant> grammar rule in ARB_vertex_program.

The grammar rules <tempVarName>, <addrVarName>, <attribVarName>, <paramArrayVarName>, <paramSingleVarName>, <resultVarName> refer to the names of temporary, address register, attribute, program parameter array, program parameter, and result variables declared in the program text.

GLX Protocol

None.

Errors

None.

New State

None.

New Implementation Dependent State

Get Value	Type	Get Command	Min Value	Description	Sec	Attrib
MAX_PROGRAM_EXEC_INSTRUCTIONS_NV	Z+	GetProgramivARB	65536	maximum program execution inst-ruction count	2.14.4.4	-
MAX_PROGRAM_CALL_DEPTH_NV	Z+	GetProgramivARB	4	maximum program call stack depth	2.14.4.4	-

(add to Table X.11. New Implementation-Dependent Values Introduced by ARB_vertex_program. Values queried by GetProgramivARB require a <pname> of VERTEX_PROGRAM_ARB.)

Revision History

```
    Rev.  Date      Author   Changes
    ----  --------  -------  ---------------------------------------------
    3     06/23/04  pbrown   Documented that vertex results are undefined
                             if the call stack overflows, and clarified that
                             RET with an empty call stack is not an error.

    2     05/16/04  pbrown   Documented terminals in modified vertex
                             program grammar.

    1     --------  pbrown   Internal pre-release revisions.
```

Name

    NV_vertex_program3

Name Strings

    GL_NV_vertex_program3

Status

    Shipping.

Version

```
    Last Modified Data:         09/27/2006
    NVIDIA Revision:            6
```

Number

    306

Dependencies

```
    ARB_vertex_program is required.
    NV_vertex_program2_option is required.
    This extension interacts with ARB_fragment_program_shadow.
```

Overview

```
    This extension, like the NV_vertex_program2_option extension,
    provides additional vertex program functionality to extend the
    standard ARB_vertex_program language and execution environment.
    ARB programs wishing to use this added functionality need only add:

        OPTION NV_vertex_program3;

    to the beginning of their vertex programs.

    New functionality provided by this extension, above and beyond that
    already provided by NV_vertex_program2_option extension, includes:

        * texture lookups in vertex programs,

        * ability to push and pop address registers on the stack,

        * address register-relative addressing for vertex attribute and
          result arrays, and

        * a second four-component condition code.
```

Issues

    Should we provided a separate "!!VP3.0" program type, like the
    "!!VP2.0" type defined in NV_vertex_program2?

```
      RESOLVED:  No.  Since ARB_vertex_program has been fully defined
      (it wasn't in the !!VP2.0 time-frame), we will simply define
```

language extensions to !!ARBvp1.0 that expose new functionality. The NV_vertex_program2_option specification followed this same pattern for the NV3X family (GeForce FX, Quadro FX).

Should this be called "NV_vertex_program3_option"?

RESOLVED: No. The similar extension to !!ARBvp1.0 called "NV_vertex_program2_option" got that name only because the simpler "NV_vertex_program2" name had already been used.

Is there a limit on the number of texture units that can be accessed by a vertex program?

RESOLVED: Yes. The limit may be lower than the total number of texture image units available and is given by the implementation-dependent constant MAX_VERTEX_TEXTURE_IMAGE_UNITS_ARB. Any program that attempts to use more unique texture image units will fail to load. Programs can use any texture image unit number, as long as they don't use too many simultaneously. As an example, the GeForce 6 series of GPUs provides 16 texture image units accessible to vertex programs, but no more than four can be used simultaneously. It is not an error to use texture image units 12-15 in a program.

This limitation is identical to the one in the ARB_vertex_shader extensions -- both extensions use the same enum to query the number of available image units. Violating this limit in GLSL results in a link error.

Is there a restriction on the texture targets that can be accessed by a vertex program?

RESOLVED: Yes -- for any texture image unit, vertex and fragment processing can not use different targets. If they do, an INVALID_OPERATION is generated at Begin-time. This resolution is consistent with resultion of the same issue in the ARB_vertex_shader extension and OpenGL 2.0.

Since vertices don't have screen space partial derivatives, how is the LOD used for texture accesses defined?

RESOLVED: The TXL instruction allows a program to explicitly set an LOD; the LOD for all other texture instructions is zero. The texture LOD bias specified in the texture object and environment do apply to all vertex texture lookups.

New Procedures and Functions

None.

New Tokens

Accepted by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

```
    MAX_VERTEX_TEXTURE_IMAGE_UNITS_ARB              0x8B4C
```

Additions to Chapter 2 of the OpenGL 1.4 Specification (OpenGL Operation)

Modify Section 2.14.2, Vertex Program Grammar and Restrictions

```
    (mostly add to existing grammar rules, as extended by
    NV_vertex_program2_option)

    <optionName>            ::= "NV_vertex_program3"

    <instruction>           ::= <TexInstruction>

    <ALUInstruction>        ::= <ASTACKop_instruction>

    <TexInstruction>        ::= <TEXop_instruction>

    <ASTACKop_instruction>  ::= <ASTACKop> <instOperandAddrVNS>

    <ASTACKop>              ::= "PUSHA"
                              | "POPA"

    <TEXop_instruction>     ::= <TEXop> <instResult> "," <instOperandV> ","
                                <texTarget>

    <TEXop>                 ::= "TEX"
                              | "TXP"
                              | "TXB"
                              | "TXL"

    <texTarget>             ::= <texImageUnit> "," <texTargetType>

    <texImageUnit>          ::= "texture" <optTexImageUnitNum>

    <optTexImageUnitNum>    ::= /* empty */
                              | "[" <texImageUnitNum> "]"

    <texImageUnitNum>       ::= <integer>
                                /*[0,MAX_TEXTURE_IMAGE_UNITS_ARB-1]*/

    <texTargetType>         ::= "1D"
                              | "2D"
                              | "3D"
                              | "CUBE"
                              | "RECT"

    <attribVtxBasic>        ::= "texcoord" "[" <arrayMemRel> "]"
                              | "attrib" "[" <arrayMemRel> "]"

    <resultVtxBasic>        ::= "texcoord" "[" <arrayMemRel> "]"
```

```
<ccMaskRule>            ::= "EQ0"
                          | "GE0"
                          | "GT0"
                          | "LE0"
                          | "LT0"
                          | "NE0"
                          | "TR0"
                          | "FL0"
                          | "EQ1"
                          | "GE1"
                          | "GT1"
                          | "LE1"
                          | "LT1"
                          | "NE1"
                          | "TR1"
                          | "FL1"
```

(modify description of reserved identifiers)

... The following strings are reserved keywords and may not be used as identifiers:

ABS, ADD, ADDRESS, ALIAS, ARA, ARL, ARR, ATTRIB, BRA, CAL, COS, DP3, DP4, DPH, DST, END, EX2, EXP, FLR, FRC, LG2, LIT, LOG, MAD, MAX, MIN, MOV, MUL, OPTION, OUTPUT, PARAM, POPA, POW, PUSHA, RCC, RCP, RET, RSQ, SEQ, SFL, SGE, SGT, SIN, SLE, SLT, SNE, SUB, SSG, STR, SWZ, TEMP, TEX, TXB, TXL, TXP, XPD, program, result, state, and vertex.

Modify Section 2.14.3.1, Vertex Attributes

(add new bindings to binding table)

Vertex Attribute Binding	Components	Underlying State
...		
vertex.texcoord[A+n]	(s,t,r,q)	indexed texture coordinate
vertex.attrib[A+n]	(x,y,z,w)	indexed generic vertex attribute

If a vertex attribute binding matches "vertex.texcoord[A+n]", where "A" is a component of an address register (Section 2.14.3.5), a texture coordinate number <c> is computed by adding the current value of the address register component and <n>. The "x", "y", "z", and "w" components of the vertex attribute variable are filled with the "s", "t", "r", and "q" components, respectively, of the vertex texture coordinates for texture unit <c>. If <c> is negative or greater than or equal to MAX_TEXTURE_COORDS_ARB, the vertex attribute variable is undefined.

If a vertex attribute binding matches "vertex.attrib[A+n]", where "A" is a component of an address register (Section 2.14.3.5), a vertex attribute number <a> is computed by adding the current value of the address register component and <n>. The "x", "y", "z", and "w" components of the vertex attribute variable are filled with the "x", "y", "z", and "w" components, respectively, of generic vertex attribute <a>. If <a> is negative or greater than or equal to MAX_VERTEX_ATTRIBS_ARB, the vertex attribute variable is undefined.

Modify Section 2.14.3.4, Vertex Program Results

(add new binding to binding table)

```
  Binding                        Components  Description
  -----------------------------  ----------  ----------------------------
  ...
  result.texcoord[A+n]           (s,t,r,q)   indexed texture coordinate
```

If a result variable binding matches "result.texcoord[A+n]", where "A" is a component of an address register (Section 2.14.3.5), a texture coordinate number <c> is computed by adding the current value of the address register component and <n>. Updates to the "x", "y", "z", and "w" components of the result variable set the "s", "t", "r" and "q" components, respectively, of the transformed vertex's texture coordinates for texture unit <c>. If <c> is negative or greater than or equal to MAX_TEXTURE_COORDS_ARB, the effects of updates to vertex attribute variable are undefined and may overwrite other programs results.

Modify Section 2.14.3.X, Condition Code Registers (added in NV_Vertex_program2_option)

The vertex program condition code registers are two four-component vectors, called CC0 and CC1. Each component of this register is one of four enumerated values: GT (greater than), EQ (equal), LT (less than), or UN (unordered). The condition code register can be used to mask writes to registers and to evaluate conditional branches.

Most vertex program instructions can optionally update one of the two condition code registers. When a vertex program instruction updates a condition code register, a condition code component is set to LT if the corresponding component of the result is less than zero, EQ if it is equal to zero, GT if it is greater than zero, and UN if it is NaN (not a number).

The condition code registers are initialized to vectors of EQ values each time a vertex program executes.

Modify Section 2.14.3.7, Vertex Program Resource Limits

(add new paragraph to end of section) In addition to the previous limits, the number of unique texture image units that can be accessed simultaneously by a vertex program is limited. The limit is given by the implementation-dependent constant MAX_VERTEX_TEXTURE_IMAGE_UNITS_ARB, and may be lower than the total number of texture image units provided. If the number of texture image units referenced by a vertex program exceeds this limit, the program will fail to load.

Modify Section 2.14.4, Vertex Program Execution Environment

(modify Begin-time error language for vertex program execution to cover invalid texture uses)

If vertex program mode is enabled and the currently bound program object does not contain a valid vertex program, the error INVALID_OPERATION will be generated by Begin, RasterPos, and any command that implicitly calls Begin (e.g., DrawArrays).

If vertex program mode is enabled and the currently bound program object accesses a texture image unit, the texture target used must be consistent with the target (if any) used for fragment processing. If vertex and fragment processing require the use of different texture targets on the same texture image unit, the error INVALID_OPERATION will be generated by Begin, RasterPos, and any command that implicitly calls Begin.

(modify instruction table) There are forty-eight vertex program instructions. Vertex program instructions may have up to eight variants, including a suffix of "C" or "C0" to allow an update of condition code register zero (section 2.14.3.X), a suffix of "C1" to allow an update of condition code register one, and a suffix of "_SAT" to clamp the result vector components to the range [0,1]. For example, the eight forms of the "ADD" instruction are "ADD", "ADDC", "ADDC0", "ADDC1", "ADD_SAT", "ADDC_SAT", "ADDC0_SAT", and "ADDC1_SAT". The instructions and their respective input and output parameters are summarized in Table X.5.

	Modifiers				
Instruction	**C**	**S**	**Inputs**	**Output**	**Description**
ABS	X	X	v	v	absolute value
ADD	X	X	v,v	v	add
ARA	X	-	a	a	address register add
ARL	X	-	s	a	address register load
ARR	X	-	v	a	address register load (round)
BRA	-	-	c	-	branch
CAL	-	-	c	-	subroutine call
COS	X	X	s	ssss	cosine
DP3	X	X	v,v	ssss	3-component dot product
DP4	X	X	v,v	ssss	4-component dot product
DPH	X	X	v,v	ssss	homogeneous dot product
DST	X	X	v,v	v	distance vector
EX2	X	X	s	ssss	exponential base 2
EXP	X	X	s	v	exponential base 2 (approximate)
FLR	X	X	v	v	floor
FRC	X	X	v	v	fraction
LG2	X	X	s	ssss	logarithm base 2
LIT	X	X	v	v	compute light coefficients
LOG	X	X	s	v	logarithm base 2 (approximate)
MAD	X	X	v,v,v	v	multiply and add
MAX	X	X	v,v	v	maximum
MIN	X	X	v,v	v	minimum
MOV	X	X	v	v	move
MUL	X	X	v,v	v	multiply
POPA	-	-	-	a	pop address register
POW	X	X	s,s	ssss	exponentiate
PUSHA	-	-	a	-	push address register
RCC	X	X	s	ssss	reciprocal (clamped)
RCP	X	X	s	ssss	reciprocal
RET	-	-	c	-	subroutine return

Instruction	Modifiers C	Modifiers S	Inputs	Output	Description
RSQ	X	X	s	ssss	reciprocal square root
SEQ	X	X	v,v	v	set on equal
SFL	X	X	v,v	v	set on false
SGE	X	X	v,v	v	set on greater than or equal
SGT	X	X	v,v	v	set on greater than
SIN	X	X	s	ssss	sine
SLE	X	X	v,v	v	set on less than or equal
SLT	X	X	v,v	v	set on less than
SNE	X	X	v,v	v	set on not equal
SSG	X	X	v	v	set sign
STR	X	X	v,v	v	set on true
SUB	X	X	v,v	v	subtract
SWZ	X	X	v	v	extended swizzle
TEX	X	X	v	v	texture lookup
TXB	X	X	v	v	texture lookup with LOD bias
TXL	X	X	v	v	texture lookup with explicit LOD
TXP	X	X	v	v	projective texture lookup
XPD	X	X	v,v	v	cross product

Table X.5: Summary of vertex program instructions. The columns "C" and "S" indicate whether the "C", "C0", and "C1" condition code update modifiers, and the "_SAT" saturation modifiers, respectively, are supported for the opcode. "v" indicates a floating-point vector input or output, "s" indicates a floating-point scalar input, "ssss" indicates a scalar output replicated across a 4-component result vector, "a" indicates a vector address register, and "c" indicates a condition code test.

Rewrite Section 2.14.4.3, Vertex Program Destination Register Update

A vertex program instruction can optionally clamp the results of a floating-point result vector to the range [0,1]. The components of the result vector are clamped to [0,1] if the saturation suffix "_SAT" is present in the instruction.

Most vertex program instructions write a 4-component result vector to a single temporary or vertex result register. Writes to individual components of the destination register are controlled by individual component write masks specified as part of the instruction.

The component write mask is specified by the <optionalMask> rule found in the <maskedDstReg> rule. If the optional mask is "", all components are enabled. Otherwise, the optional mask names the individual components to enable. The characters "x", "y", "z", and "w" match the x, y, z, and w components respectively. For example, an optional mask of ".xzw" indicates that the x, z, and w components should be enabled for writing but the y component should not. The grammar requires that the destination register mask components must be listed in "xyzw" order. The condition code write mask is specified by the <ccMask> rule found in the <instResultCC> and <instResultAddrCC> rules. Otherwise, the selected condition code register is loaded and swizzled according to the swizzle codes specified by <swizzleSuffix>. Each component of the swizzled condition code is tested according to the rule given by <ccMaskRule>.

<ccMaskRule> may have the values "EQ", "NE", "LT", "GE", LE", or "GT", which mean to enable writes if the corresponding condition code field evaluates to equal, not equal, less than, greater than or equal, less than or equal, or greater than, respectively. Comparisons involving condition codes of "UN" (unordered) evaluate to true for "NE" and false otherwise. For example, if the condition code is (GT,LT,EQ,GT) and the condition code mask is "(NE.zyxw)", the swizzle operation will load (EQ,LT,GT,GT) and the mask will thus will enable writes on the y, z, and w components. In addition, "TR" always enables writes and "FL" always disables writes, regardless of the condition code. If the condition code mask is empty, it is treated as "(TR)".

Each component of the destination register is updated with the result of the vertex program instruction if and only if the component is enabled for writes by both the component write mask and the condition code write mask. Otherwise, the component of the destination register remains unchanged.

A vertex program instruction can also optionally update the condition code register. The condition code is updated if the condition code register update suffix "C" is present in the instruction. The instruction "ADDC" will update the condition code; the otherwise equivalent instruction "ADD" will not. If condition code updates are enabled, each component of the destination register enabled for writes is compared to zero. The corresponding component of the condition code is set to "LT", "EQ", or "GT", if the written component is less than, equal to, or greater than zero, respectively. Condition code components are set to "UN" if the written component is NaN (not a number). Values of -0.0 and +0.0 both evaluate to "EQ". If a component of the destination register is not enabled for writes, the corresponding condition code component is also unchanged.

In the following example code,

```
# R1=(-2, 0, 2, NaN)              R0                 CC
MOVC R0, R1;               # ( -2,  0,   2, NaN) (LT,EQ,GT,UN)
MOVC R0.xyz, R1.yzwx;      # (  0,  2, NaN, NaN) (EQ,GT,UN,UN)
MOVC R0 (NE), R1.zywx;     # (  0,  0, NaN,  -2) (EQ,EQ,UN,LT)
```

the first instruction writes (-2,0,2,NaN) to R0 and updates the condition code to (LT,EQ,GT,UN). The second instruction, only the "x", "y", and "z" components of R0 and the condition code are updated, so R0 ends up with (0,2,NaN,NaN) and the condition code ends up with (EQ,GT,UN,UN). In the third instruction, the condition code mask disables writes to the x component (its condition code field is "EQ"), so R0 ends up with (0,0,NaN,-2) and the condition code ends up with (EQ,EQ,UN,LT).

The following pseudocode illustrates the process of writing a result vector to the destination register. In the pseudocode, "instrSaturate" is TRUE if and only if result saturation is enabled, "instrMask" refers to the component write mask given by the <optWriteMask> rule. "ccMaskRule" refers to the condition code mask rule given by <ccMask> and "updatecc" is TRUE if and only if condition code updates are enabled. "result", "destination", and "cc" refer to the result vector, the register selected by <dstRegister>

    and the condition code, respectively. Condition codes do not exist
    in the VP1 execution environment.

```
      boolean TestCC(CondCode field) {
          switch (ccMaskRule) {
          case "EQ":  return (field == "EQ");
          case "NE":  return (field != "EQ");
          case "LT":  return (field == "LT");
          case "GE":  return (field == "GT" || field == "EQ");
          case "LE":  return (field == "LT" || field == "EQ");
          case "GT":  return (field == "GT");
          case "TR":  return TRUE;
          case "FL":  return FALSE;
          case "":    return TRUE;
          }
      }

      enum GenerateCC(float value) {
        if (value == NaN) {
          return UN;
        } else if (value < 0) {
          return LT;
        } else if (value == 0) {
          return EQ;
        } else {
          return GT;
        }
      }
      void UpdateDestination(floatVec destination, floatVec result)
      {
          floatVec merged;
          ccVec    mergedCC;

          // Clamp result components to [0,1] if requested in the instruction.
          if (instrSaturate) {
              if (result.x < 0)      result.x = 0;
              else if (result.x > 1) result.x = 1;
              if (result.y < 0)      result.y = 0;
              else if (result.y > 1) result.y = 1;
              if (result.z < 0)      result.z = 0;
              else if (result.z > 1) result.z = 1;
              if (result.w < 0)      result.w = 0;
              else if (result.w > 1) result.w = 1;
          }

          // Merge the converted result into the destination register, under
          // control of the compile- and run-time write masks.
          merged = destination;
          mergedCC = cc;
          if (instrMask.x && TestCC(cc.c***)) {
              merged.x = result.x;
              if (updatecc) mergedCC.x = GenerateCC(result.x);
          }
          if (instrMask.y && TestCC(cc.*c**)) {
              merged.y = result.y;
              if (updatecc) mergedCC.y = GenerateCC(result.y);
          }
          if (instrMask.z && TestCC(cc.**c*)) {
              merged.z = result.z;
              if (updatecc) mergedCC.z = GenerateCC(result.z);
          }
          if (instrMask.w && TestCC(cc.***c)) {
              merged.w = result.w;
              if (updatecc) mergedCC.w = GenerateCC(result.w);
          }

          // Write out the new destination register and condition code.
          destination = merged;
          cc = mergedCC;
      }
```

While this rule describes floating-point results, the same logic applies to the integer results generated by the ARA, ARL, and ARR instructions.

Add to Section 2.14.4.5, Vertex Program Options

Section 2.14.4.5.3, NV_vertex_program3 Program Option

If a vertex program specifies the "NV_vertex_program3" option, the ARB_vertex_program grammar and execution environment are extended to take advantage of all the features of the "NV_vertex_program2" option, plus the following features:

* several new instructions:

  * POPA -- pop address register off stack
  * PUSHA -- push address register onto stack
  * TEX -- texture lookup
  * TXB -- texture lookup w/LOD bias
  * TXL -- texture lookup w/explicit LOD
  * TXP -- projective texture lookup

* address register-relative addressing for vertex texture coordinate and generic attribute arrays,

* address register-relative addressing for vertex texture coordinate result array, and

* a second four-component condition code.

Modify Section 2.14.5.34, RET: Subroutine Call Return

The RET instruction conditionally returns from a subroutine initiated by a CAL instruction by popping an instruction reference off the top of the call stack and transferring control to the referenced instruction. The following pseudocode describes the operation of the instruction:

```
  if (TestCC(cc.c***) || TestCC(cc.*c**) ||
      TestCC(cc.**c*) || TestCC(cc.***c)) {
    if (callStackDepth <= 0) {
      // terminate vertex program normally
    } else {
      callStackDepth--;
      if (callStack[callStackDepth] is a instruction reference) {
        instruction = callStack[callStackDepth];
      } else {
        // terminate vertex program abnormally
      }
    }

    // continue execution at <instruction>
  } else {
    // do nothing
  }
```

In the pseudocode, <callStackDepth> is the depth of the call stack, <callStack> is an array holding the call stack, and <instruction> is a reference to an instruction previously pushed onto the call stack.

If the call stack is empty when RET executes, the vertex program terminates normally.

The vertex program terminates abnormally if the entry at the top of the call stack is not an instruction reference pushed by CAL. When a vertex program terminates abnormally, all of the vertex program results are undefined.

Add to Section 2.14.5, Vertex Program Instruction Set

Section 2.14.5.43, POPA: Pop Address Register Stack

The POPA instruction generates a integer result vector by popping an entry off of the call stack.

```
  if (callStackDepth <= 0) {
    terminate vertex program;
  } else {
    callStackDepth--;
    if (callStack[callStackDepth] is an address register) {
      iresult = callStack[callStackDepth];
    } else {
      terminate vertex program;
    }
  }
```

In the pseudocode, <callStackDepth> is the current depth of the call stack and <callStack> is an array holding the call stack.

The vertex program terminates abnormally if it executes a POPA instruction when the call stack is empty, or when the entry at the top of the call stack is not an address register pushed by PUSHA. When a vertex program terminates abnormally, all of the vertex program results are undefined.

Section 2.14.5.44, PUSHA: Push Address Register Stack

The PUSHA instruction pushes the address register operand onto the call stack, which is also used for subroutine calls. The PUSHA instruction does not generate a result vector.

```
  tmp = AddrVectorLoad(op0);
  if (callStackDepth >= MAX_PROGRAM_CALL_DEPTH_NV) {
    terminate vertex program;
  } else {
    callStack[callStackDepth] = tmp;
    callStackDepth++;
  }
```

In the pseudocode, <callStackDepth> is the current depth of the call stack and <callStack> is an array holding the call stack.

The vertex program terminates abnormally if it executes a PUSHA instruction when the call stack is full. When a vertex program terminates abnormally, all of the vertex program results are undefined.

Component swizzling is not supported when the operand is loaded.

Section 2.14.5.45, TEX: Texture Lookup

The TEX instruction uses the single vector operand to perform a lookup in the specified texture map, yielding a 4-component result vector containing filtered texel values. The (s,t,r,q) coordinates used for the texture lookup are (x,y,z,1), where x, y, and z are components of the vector operand.

```
  tmp = VectorLoad(op0);
  result = TextureSample(tmp.x, tmp.y, tmp.z, 1.0, 0.0, unit, target);
```

where <unit> and <target> are the texture image unit number and target type, matching the <texImageUnitNum> and <texTargetType> grammar rules.

The resulting sample is mapped to RGBA as described in Table 3.21, and the R, G, B, and A values are written to the x, y, z, and w components, respectively, of the result vector.

Since partial derivatives of the texture coordinates are not defined, the base LOD value for vertex texture lookups is defined to be zero. The value of lambda' used in equation 3.16 will be simply clamp(texobj_bias + texunit_bias).

Section 2.14.5.46, TXB: Texture Lookup (With LOD Bias)

The TXB instruction uses the single vector operand to perform a lookup in the specified texture map, yielding a 4-component result vector containing filtered texel values. The (s,t,r,q) coordinates used for the texture lookup are (x,y,z,1), where x, y, and z are components of the vector operand. The w component of the operand is used as an additional LOD bias.

```
  tmp = VectorLoad(op0);
  result = TextureSample(tmp.x, tmp.y, tmp.z, 1.0, tmp.w, unit, target);
```

where <unit> and <target> are the texture image unit number and target type, matching the <texImageUnitNum> and <texTargetType> grammar rules.

The resulting sample is mapped to RGBA as described in Table 3.21, and the R, G, B, and A values are written to the x, y, z, and w components, respectively, of the result vector.

Since partial derivatives of the texture coordinates are not defined, the base LOD value for vertex texture lookups is defined to be zero. The value of lambda' used in equation 3.16 will be simply clamp(texobj_bias + texunit_bias + tmp.w).

Since the base LOD value is zero, the TXB instruction is completely equivalent to the TXL instruction, where the w component contains an explicit base LOD value.

Section 2.14.5.47, TXL: Texture Lookup (With Explicit LOD)

The TXL instruction uses the single vector operand to perform a lookup in the specified texture map, yielding a 4-component result vector containing filtered texel values. The (s,t,r,q) coordinates used for the texture lookup are (x,y,z,1), where x, y, and z are components of the vector operand. The w component of the operand is used as the base LOD for the texture lookup.

```
  tmp = VectorLoad(op0);
  result = TextureSampleLOD(tmp.x, tmp.y, tmp.z, 1.0, tmp.w, unit, target);
```

where <unit> and <target> are the texture image unit number and target type, matching the <texImageUnitNum> and <texTargetType> grammar rules.

The resulting sample is mapped to RGBA as described in Table 3.21, and the R, G, B, and A values are written to the x, y, z, and w components, respectively, of the result vector.

The value of lambda' used in equation 3.16 will be simply tmp.w + clamp(texobj_bias + texunit_bias), where tmp.w is the base LOD.

Section 2.14.5.48, TXP: Texture Lookup (Projective)

The TXP instruction uses the single vector operand to perform a lookup in the specified texture map, yielding a 4-component result vector containing filtered texel values. The (s,t,r,q) coordinates used for the texture lookup are (x,y,z,w), where x, y, z, and w are the four components of the vector operand.

```
  tmp = VectorLoad(op0);
  result = TextureSample(tmp.x, tmp.y, tmp.z, tmp.w, 0.0, unit, target);
```

where <unit> and <target> are the texture image unit number and target type, matching the <texImageUnitNum> and <texTargetType> grammar rules.

The resulting sample is mapped to RGBA as described in Table 3.21, and the R, G, B, and A values are written to the x, y, z, and w components, respectively, of the result vector.

Since partial derivatives of the texture coordinates are not defined, the base LOD value for vertex texture lookups is defined to be zero. The value of lambda' used in equation 3.16 will be simply clamp(texobj_bias + texunit_bias).

Additions to Chapter 3 of the OpenGL 1.4 Specification (Rasterization)

None.

Additions to Chapter 4 of the OpenGL 1.4 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 1.4 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 1.4 Specification (State and State Requests)

None.

Additions to Appendix A of the OpenGL 1.4 Specification (Invariance)

None.

Additions to the AGL/GLX/WGL Specifications

None.

Dependencies on ARB_vertex_program

ARB_vertex_program is required.

This specification and NV_vertex_program2_option are based on a modified version of the grammar published in the ARB_vertex_program specification. This modified grammar includes a few structural changes to better accommodate new functionality from this and other extensions, but should be functionally equivalent to the ARB_vertex_program grammar. See NV_vertex_program2_option for details on the base grammar.

Dependencies on NV_vertex_program2_option

NV_vertex_program2_option is required.

If the NV_vertex_program3 program option is specified, all the functionality described in both this extension and the NV_vertex_program2_option specification is available.

Dependencies on ARB_fragment_program_shadow

If this extension and ARB_fragment_program shadow are both supported, vertex programs may include the option statement:

```
OPTION ARB_fragment_program_shadow;
```

which enables the use of the SHADOW1D and SHADOW2D texture targets in texture lookup instructions, as described in the ARB_fragment_program_shadow specification.

NVIDIA NOTE: Drivers prior to September 2006 do not support the use of this option, and will not accept texture lookups with SHADOW1D and SHADOW2D targets. Shadow mapping in vertex programs will result in software fallbacks on GeForce 6 and GeForce 7 series GPUs, but may be done in hardware on future GPUs.

Errors

None.

New State

None.

New Implementation Dependent State:

Get Value	Type	Get Command	Minimum Value	Description	Section	Attr.
MAX_VERTEX_TEXTURE_ IMAGE_UNITS_ARB	Z+	GetIntegerv	1	Number of separate texture image units that can be accessed by a vertex program	2.14.3.7	-

Revision History

```
    Rev.    Date     Author    Changes
    ----  --------  --------  -----------------------------------------
    6     09/27/06  pbrown    Document that ARB_fragment_program_shadow is
                              allowed, to enable the use of "SHADOW1D" and
                              "SHADOW2D" targets for texture lookups.

    5     11/07/05  pbrown    Fix PUSHA documentation to specify the right
                              constant name used for overflow testing.

    4     09/01/05  pbrown    Fix spec language to document that a vertex
                              program will fail to compile if it uses "too
                              many" textures -- previously only documented
                              in the issues section.

    3     08/25/05  pbrown    Document that using a different texture target
                              than fragment processing on the same texture
                              unit results in an INVALID_OPERATION error at
                              Begin time.  This is consistent with GLSL
                              language in the ARB_shader_objects and OpenGL
                              2.0 specifications.  The implementation has
```

Name

    NV_vertex_program4

Name Strings

    (none)

Contact

    Pat Brown, NVIDIA Corporation (pbrown 'at' nvidia.com)

Status

    Shipping for GeForce 8 Series (November 2006)

Version

    Last Modified Date: 10/06/06
    NVIDIA Revision: 5

Number

    325

Dependencies

    OpenGL 1.1 is required.

    This extension is written against the OpenGL 2.0 specification.

    ARB_vertex_program is required.

    NV_gpu_program4 is required. This extension is supported if
    "GL_NV_gpu_program4" is found in the extension string.

    NVX_instanced_arrays affects the definition of this extension.

Overview

    This extension builds on the common assembly instruction set
    infrastructure provided by NV_gpu_program4, adding vertex program-specific
    features.

    This extension provides the ability to specify integer vertex attributes
    that are passed to vertex programs using integer data types, rather than
    being converted to floating-point values as in existing vertex attribute
    functions. The set of input and output bindings provided includes all
    bindings supported by ARB_vertex_program. This extension provides
    additional input bindings identifying the index of the vertex when vertex
    arrays are used ("vertex.id") and the instance number when instanced
    arrays are used ("vertex.instance", requires EXT_draw_instanced). It
    also provides output bindings allowing vertex programs to directly specify
    clip distances (for user clipping) plus a set of generic attributes that
    allow programs to pass a greater number of attributes to subsequent
    pipeline stages than is possible using only the pre-defined fixed-function
    vertex outputs.

By and large, programs written to ARB_vertex_program can be ported directly by simply changing the program header from "!!ARBvp1.0" to "!!NVvp4.0", and then modifying instructions to take advantage of the expanded feature set. There are a small number of areas where this extension is not a functional superset of previous vertex program extensions, which are documented in the NV_gpu_program4 specification.

New Procedures and Functions

```
    void VertexAttribI1iEXT(uint index, int x);
    void VertexAttribI2iEXT(uint index, int x, int y);
    void VertexAttribI3iEXT(uint index, int x, int y, int z);
    void VertexAttribI4iEXT(uint index, int x, int y, int z, int w);

    void VertexAttribI1uiEXT(uint index, uint x);
    void VertexAttribI2uiEXT(uint index, uint x, uint y);
    void VertexAttribI3uiEXT(uint index, uint x, uint y, uint z);
    void VertexAttribI4uiEXT(uint index, uint x, uint y, uint z, uint w);

    void VertexAttribI1ivEXT(uint index, const int *v);
    void VertexAttribI2ivEXT(uint index, const int *v);
    void VertexAttribI3ivEXT(uint index, const int *v);
    void VertexAttribI4ivEXT(uint index, const int *v);

    void VertexAttribI1uivEXT(uint index, const uint *v);
    void VertexAttribI2uivEXT(uint index, const uint *v);
    void VertexAttribI3uivEXT(uint index, const uint *v);
    void VertexAttribI4uivEXT(uint index, const uint *v);

    void VertexAttribI4bvEXT(uint index, const byte *v);
    void VertexAttribI4svEXT(uint index, const short *v);
    void VertexAttribI4ubvEXT(uint index, const ubyte *v);
    void VertexAttribI4usvEXT(uint index, const ushort *v);

    void VertexAttribIPointerEXT(uint index, int size, enum type,
                                 sizei stride, const void *pointer);

    void GetVertexAttribIivEXT(uint index, enum pname, int *params);
    void GetVertexAttribIuivEXT(uint index, enum pname, uint *params);
```

(note: all these functions are shared with the EXT_gpu_shader4 extension.)

New Tokens

Accepted by the <pname> parameters of GetVertexAttribdv, GetVertexAttribfv, GetVertexAttribiv, GetVertexAttribIivEXT, and GetVertexAttribIuivEXT:

```
      VERTEX_ATTRIB_ARRAY_INTEGER_EXT                   0x88FD
```

(note: this token is shared with the EXT_gpu_shader4 extension.)

Additions to Chapter 2 of the OpenGL 2.0 Specification (OpenGL Operation)

Modify Section 2.7 (Vertex Specification), p.20

(insert before last paragraph, p.22) The commands

```
  void VertexAttribI[1234]{i,ui}EXT(uint index, T values);
  void VertexAttribI[1234]{i,ui}vEXT(uint index, T values);
  void VertexAttribI4{b,s,ub,us}vEXT(uint index, T values);
```

specify fixed-point coordinates that are not converted to floating-point values, but instead are represented as signed or unsigned integer values. Vertex programs that use integer instructions may read these attributes using integer data types. A vertex program that attempts to read a vertex attribute as a float will get undefined results if the attribute was specified as an integer, and vice versa.

(modify second paragraph, p.23) Setting generic vertex attribute zero specifies a vertex; the four vertex coordinates are taken from the values of attribute zero. A Vertex2, Vertex3, or Vertex4 command is completely equivalent to the corresponding VertexAttrib* or VertexAttribI* command with an index of zero. ...

(insert at end of function list, p.24)

```
  void VertexAttribIPointerEXT(uint index, int size, enum type,
                               sizei stride, const void *pointer);
```

(modify last paragraph, p.24) The <index> parameter in the VertexAttribPointer and VertexAttribIPointerEXT commands identify the generic vertex attribute array being described. The error INVALID_VALUE is generated if <index> is greater than or equal to MAX_VERTEX_ATTRIBS. Generic attribute arrays with integer <type> arguments can be handled in one of three ways: converted to float by normalizing to [0,1] or [-1,1] as specified in table 2.9, converted directly to float, or left as integer values. Data for an array specified by VertexAttribPointer will be converted to floating-point by normalizing if the <normalized> parameter is TRUE, and converted directly to floating-point otherwise. Data for an array specified by VertexAttribIPointerEXT will always be left as integer values.

(modify Table 2.4, p. 25)

Command	Sizes	Integer Handling	Types
VertexPointer	2,3,4	cast	...
NormalPointer	3	normalize	...
ColorPointer	3,4	normalize	...
SecondaryColorPointer	3	normalize	...
IndexPointer	1	cast	...
FogCoordPointer	1	n/a	...
TexCoordPointer	1,2,3,4	cast	...
EdgeFlagPointer	1	integer	...
VertexAttribPointer	1,2,3,4	flag	...
VertexAttribIPointerEXT	1,2,3,4	integer	byte, ubyte, short, ushort, int, uint

Table 2.4: Vertex array sizes (values per vertex) and data types. The "integer handling" column indicates how fixed-point data types are handled: "cast" means that they converted to floating-point directly, "normalize" means that they are converted to floating-point by normalizing to [0,1] (for unsigned types) or [-1,1] (for signed types), "integer" means that they remain as integer values, and "flag" means that either "cast" or "normalized" applies, depending on the setting of the <normalized> flag in VertexAttribPointer.

(modify end of pseudo-code, pp. 27-28)

```
for (j = 1; j < genericAttributes; j++) {
  if (generic vertex attribute j array enabled) {
    if (generic vertex attribute j array is a pure integer array) {
      VertexAttribI[size][type]vEXT(j, generic vertex attribute j
                                    array element i);
    } else if (generic vertex attribute j array normalization flag
               is set and <type> is not FLOAT or DOUBLE) {
      VertexAttrib[size]N[type]v(j, generic vertex attribute j
                                 array element i);
    } else {
      VertexAttrib[size][type]v(j, generic vertex attribute j
                                array element i);
    }
  }
}

if (generic vertex attribute 0 array enabled) {
  if (generic vertex attribute 0 array is a pure integer array) {
    VertexAttribI[size][type]vEXT(0, generic vertex attribute 0
                                  array element i);
  } else if (generic vertex attribute 0 array normalization flag
             is set and <type> is not FLOAT or DOUBLE) {
    VertexAttrib[size]N[type]v(0, generic vertex attribute 0
                               array element i);
  } else {
    VertexAttrib[size][type]v(0, generic vertex attribute 0
                              array element i);
  }
}
```

Modify Section 2.X, GPU Programs

(insert after second paragraph)

Vertex Programs

Vertex programs are used to compute the transformed attributes of a vertex, in lieu of the set of fixed-function operations described in sections 2.10 through 2.13. Vertex programs are run on a single vertex at a time, and the state of neighboring vertices is not available. The inputs available to a vertex program are the vertex attributes described in section 2.7. The results of the program are the attributes of a transformed vertex, which include (among other things) a transformed position, colors, and texture coordinates. The vertices transformed by a vertex program are then processed normally by the remainder of the GL pipeline.

Modify Section 2.X.2, Program Grammar

(replace third paragraph)

Vertex programs are required to begin with the header string "!!NVvp4.0". This header string identifies the subsequent program body as being a vertex program and indicates that it should be parsed according to the base NV_gpu_program4 grammar plus the additions below. Program string parsing begins with the character immediately following the header string.

(add the following grammar rules to the NV_gpu_program4 base grammar)

```
<resultUseW>            ::= <resultVarName> <arrayMem> <optWriteMask>
                          | <resultColor> <optWriteMask>
                          | <resultColor> "." <colorType> <optWriteMask>
                          | <resultColor> "." <faceType> <optWriteMask>
                          | <resultColor> "." <faceType> "." <colorType>
                            "." <optWriteMask>

<resultUseD>            ::= <resultColor>
                          | <resultColor> "." <colorType>
                          | <resultColor> "." <faceType>
                          | <resultColor> "." <faceType> "." <colorType>
                          | <resultMulti>

<attribBasic>           ::= <vtxPrefix> "position"
                          | <vtxPrefix> "weight" <optArrayMemAbs>
                          | <vtxPrefix> "normal"
                          | <vtxPrefix> "fogcoord"
                          | <attribTexCoord> <optArrayMemAbs>
                          | <attribGeneric> <arrayMemAbs>
                          | <vtxPrefix> "id"
                          | <vtxPrefix> "instance"

<attribColor>           ::= <vtxPrefix> "color"

<attribMulti>           ::= <attribTexCoord> <arrayRange>
                          | <attribGeneric> <arrayRange>
```

```
    <attribTexCoord>        ::= <vtxPrefix> "texcoord"

    <attribGeneric>         ::= <vtxPrefix> "attrib"

    <vtxPrefix>             ::= "vertex" "."

    <resultBasic>           ::= <resPrefix> "position"
                              | <resPrefix> "fogcoord"
                              | <resPrefix> "pointsize"
                              | <resultTexCoord> <optArrayMemAbs>
                              | <resultClip> <arrayMemAbs>
                              | <resultGeneric> <arrayMemAbs>
                              | <resPrefix> "id"

    <resultColor>           ::= <resPrefix> "color"

    <resultMulti>           ::= <resultTexCoord> <arrayRange>
                              | <resultClip> <arrayRange>
                              | <resultGeneric> <arrayRange>

    <resultTexCoord>        ::= <resPrefix> "texcoord"

    <resultClip>            ::= <resPrefix> "clip"

    <resultGeneric>         ::= <resPrefix> "attrib"

    <resPrefix>             ::= "result" "."
```

(add the following subsection to Section 2.X.3.2, Program Attribute Variables)

Vertex program attribute variables describe the attributes of the vertex being transformed, as specified by the application. The set of available bindings is enumerated in Table X.X. Except where otherwise noted, all vertex program attribute bindings are four-component floating-point vectors.

Vertex Attribute Binding	Components	Underlying State
vertex.position	(x,y,z,w)	object coordinates
vertex.normal	(x,y,z,1)	normal
vertex.color	(r,g,b,a)	primary color
vertex.color.primary	(r,g,b,a)	primary color
vertex.color.secondary	(r,g,b,a)	secondary color
vertex.fogcoord	(f,0,0,1)	fog coordinate
vertex.texcoord	(s,t,r,q)	texture coordinate, unit 0
vertex.texcoord[n]	(s,t,r,q)	texture coordinate, unit n
vertex.attrib[n]	(x,y,z,w)	generic vertex attribute n
vertex.id	(id,-,-,-)	vertex identifier (integer)
vertex.instance	(i,-,-,-)	primitive instance number (integer)
vertex.texcoord[n..o]	(x,y,z,w)	array of texture coordinates
vertex.attrib[n..o]	(x,y,z,w)	array of generic vertex attributes

Table X.X, Vertex Program Attribute Bindings. <n> and <o> refer to integer constants. Only the "vertex.texcoord" and "vertex.attrib" bindings are available in arrays.

NVIDIA Note: The "vertex.weight" and "vertex.matrixindex" bindings described in ARB_vertex_program use state provided only by extensions not supported by NVIDIA implementations and are not available.

If a vertex attribute binding matches "vertex.position", the "x", "y", "z" and "w" components of the vertex attribute variable are filled with the "x", "y", "z", and "w" components, respectively, of the vertex position.

If a vertex attribute binding matches "vertex.normal", the "x", "y", and "z" components of the vertex attribute variable are filled with the "x", "y", and "z" components, respectively, of the vertex normal. The "w" component is filled with 1.

If a vertex attribute binding matches "vertex.color" or "vertex.color.primary", the "x", "y", "z", and "w" components of the vertex attribute variable are filled with the "r", "g", "b", and "a" components, respectively, of the vertex color.

If a vertex attribute binding matches "vertex.color.secondary", the "x", "y", "z", and "w" components of the vertex attribute variable are filled with the "r", "g", "b", and "a" components, respectively, of the vertex secondary color.

If a vertex attribute binding matches "vertex.fogcoord", the "x" component of the vertex attribute variable is filled with the vertex fog coordinate. The "y", "z", and "w" coordinates are filled with 0, 0, and 1, respectively.

If a vertex attribute binding matches "vertex.texcoord" or "vertex.texcoord[n]", the "x", "y", "z", and "w" components of the vertex attribute variable are filled with the "s", "t", "r", and "q" components, respectively, of the vertex texture coordinate set <n>. If "[n]" is omitted, texture coordinate set zero is used.

If a vertex attribute binding matches "vertex.instance", the "x" component of the vertex attribute variable is filled with the integer instance number for the primitive to which the vertex belongs. The "y", "z", and "w" components are undefined.

If a vertex attribute binding matches "vertex.attrib[n]", the "x", "y", "z" and "w" components of the generic vertex attribute variable are filled with the "x", "y", "z", and "w" components, respectively, of generic vertex attribute <n>. Note that "vertex.attrib[0]" and "vertex.position" are equivalent. Generic vertex attribute bindings are typeless, and can be interpreted as having floating-point, signed integer, or unsigned integer values, depending on how they are used in the program text. If a vertex attribute is read using a data type different from the one used to specify the generic attribute, the values corresponding to the binding are undefined.

As described in section 2.7, setting a generic vertex attribute may leave a corresponding conventional vertex attribute undefined, and vice versa. To prevent inadvertent use of attribute pairs with undefined attributes, a vertex program will fail to load if it binds both a conventional vertex attribute and a generic vertex attribute listed in the same row of Table X.X.

Conventional Attribute Binding	Generic Attribute Binding
vertex.position	vertex.attrib[0]
vertex.normal	vertex.attrib[2]
vertex.color	vertex.attrib[3]
vertex.color.primary	vertex.attrib[3]
vertex.color.secondary	vertex.attrib[4]
vertex.fogcoord	vertex.attrib[5]
vertex.texcoord	vertex.attrib[8]
vertex.texcoord[0]	vertex.attrib[8]
vertex.texcoord[1]	vertex.attrib[9]
vertex.texcoord[2]	vertex.attrib[10]
vertex.texcoord[3]	vertex.attrib[11]
vertex.texcoord[4]	vertex.attrib[12]
vertex.texcoord[5]	vertex.attrib[13]
vertex.texcoord[6]	vertex.attrib[14]
vertex.texcoord[7]	vertex.attrib[15]
vertex.texcoord[n]	vertex.attrib[8+n]

Table X.X: Invalid Vertex Attribute Binding Pairs. Vertex programs may not bind both attributes listed in any row. The <n> in the last row matches the number of any valid texture unit.

If a vertex attribute binding matches "vertex.texcoord[n..o]" or "vertex.attrib[n..o]", a sequence of 1+<o>-<n> texture coordinate bindings are created. For texture coordinates, it is as though the sequence "vertex.texcoord[n], vertex.texcoord[n+1], ... vertex.texcoord[o]" were specfied. These bindings are available only in explicit declarations of array variables. A program will fail to load if <n> is greater than <o>.

When doing vertex array rendering using buffer objects, a vertex ID is also available. If a vertex attribute binding matches "vertex.id", the "x" component of this vertex attribute is filled with the integer index <i> implicitly passed to ArrayElement() to specify the vertex. The vertex ID is defined if and only if:

* the vertex comes from a vertex array command that specifies a complete primitive (e.g., DrawArrays, DrawElements),

* all enabled vertex arrays have non-zero buffer object bindings, and

* the vertex does not come from a display list (even if the display list was compiled using DrawArrays/DrawElements using buffer objects).

The "y", "z", and "w" components of the vertex attribute are always undefined.

(add the following subsection to section 2.X.3.5, Program Results.)

Vertex programs produce vertices, and the set of result variables available to such programs correspond to the attributes of a transformed vertex. The set of allowable result variable bindings for vertex and fragment programs is given in Table X.4.

Binding	Components	Description
result.position	(x,y,z,w)	position in clip coordinates
result.color	(r,g,b,a)	front-facing primary color
result.color.primary	(r,g,b,a)	front-facing primary color
result.color.secondary	(r,g,b,a)	front-facing secondary color
result.color.front	(r,g,b,a)	front-facing primary color
result.color.front.primary	(r,g,b,a)	front-facing primary color
result.color.front.secondary	(r,g,b,a)	front-facing secondary color
result.color.back	(r,g,b,a)	back-facing primary color
result.color.back.primary	(r,g,b,a)	back-facing primary color
result.color.back.secondary	(r,g,b,a)	back-facing secondary color
result.fogcoord	(f,*,*,*)	fog coordinate
result.pointsize	(s,*,*,*)	point size
result.texcoord	(s,t,r,q)	texture coordinate, unit 0
result.texcoord[n]	(s,t,r,q)	texture coordinate, unit n
result.attrib[n]	(x,y,z,w)	generic interpolant n
result.clip[n]	(d,*,*,*)	clip plane distance
result.texcoord[n..o]	(s,t,r,q)	texture coordinates n thru o
result.attrib[n..o]	(x,y,z,w)	generic interpolants n thru o
result.clip[n..o]	(d,*,*,*)	clip distances n thru o
result.id	(id,*,*,*)	vertex id

Table X.4: Vertex Program Result Variable Bindings. Components labeled "*" are unused.

If a result variable binding matches "result.position", updates to the "x", "y", "z", and "w" components of the result variable modify the "x", "y", "z", and "w" components, respectively, of the transformed vertex's clip coordinates. Final window coordinates will be generated for the vertex as described in section 2.14.4.4.

If a result variable binding match begins with "result.color", updates to the "x", "y", "z", and "w" components of the result variable modify the "r", "g", "b", and "a" components, respectively, of the corresponding vertex color attribute in Table X.4. Color bindings that do not specify "front" or "back" are consided to refer to front-facing colors. Color bindings that do not specify "primary" or "secondary" are considered to refer to primary colors.

If a result variable binding matches "result.fogcoord", updates to the "x" component of the result variable set the transformed vertex's fog coordinate. Updates to the "y", "z", and "w" components of the result variable have no effect.

If a result variable binding matches "result.pointsize", updates to the "x" component of the result variable set the transformed vertex's point size. Updates to the "y", "z", and "w" components of the result variable have no effect.

If a result variable binding matches "result.texcoord" or "result.texcoord[n]", updates to the "x", "y", "z", and "w" components of the result variable set the "s", "t", "r" and "q" components, respectively, of the transformed vertex's texture coordinates for texture unit <n>. If "[n]" is omitted, texture unit zero is selected.

If a result variable binding matches "result.attrib[n]", updates to the "x", "y", "z", and "w" components of the result variable set the "x", "y", "z", and "w" components of the generic interpolant <n>. Generic interpolants may be used by subsequent geometry or fragment program invocations, but are not available to fixed-function fragment processing.

If a result variable binding matches "result.clip[n]", updates to the "x" component of the result variable set the clip distance for clip plane <n>.

If a result variable binding matches "result.texcoord[n..o]", "result.attrib[n..o]", or "result.clip[n..o]", a sequence of 1+<o>-<n> bindings is created. For texture coordinates, it is as though the sequence "result.texcoord[n], result.texcoord[n+1], ... result.texcoord[o]" were specfied. This binding is available only in explicit declarations of array variables. A program will fail to load if <n> is greater than <o>.

If a result variable binding matches "result.id", updates to the "x" component of the result variable provide a integer vertex identifier available to geometry programs using the "vertex[m].id" attribute binding. If a geometry program using vertex IDs is active and a vertex program is active, the vertex program must write "result.id" or the vertex ID number is undefined.

(add the following subsection to section 2.X.5, Program Options.)

Section 2.X.5.Y, Vertex Program Options

+ Position-Invariant Vertex Programs (ARB_position_invariant)

If a vertex program specifies the "ARB_position_invariant" option, the program is used to generate all transformed vertex attributes except for position. Instead, clip coordinates are computed as specified in section 2.10. Additionally, user clipping is performed as described in section 2.11. Use of position-invariant vertex programs should generally guarantee that the transformed position of a vertex should be the same whether vertex program mode is enabled or disabled, allowing for correct mixed multi-pass rendering semantics.

When the position-invariant option is specified in a vertex program, vertex programs can no longer declare (explicitly or implicitly) a result variable bound to "result.position". A semantic restriction is added to indicate that a vertex program will fail to load if the number of instructions it contains exceeds the implementation-dependent limit minus four.

(add the following subsection to section 2.X.6, Program Declarations.)

Section 2.X.6.1, Vertex Program Declarations

No declarations are supported at present for vertex programs.

Additions to Chapter 3 of the OpenGL 2.0 Specification (Rasterization)

None.

Additions to Chapter 4 of the OpenGL 2.0 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 2.0 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 2.0 Specification (State and State Requests)

Modify Section 6.1.14, Shader and Program Queries (p. 256)

(modify 2nd paragraph, p.259) The commands

```
  ...
  void GetVertexAttribIivEXT(uint index, enum pname, int *params);
  void GetVertexAttribIuivEXT(uint index, enum pname, uint *params);
```

obtain the... <pname> must be one of VERTEX_ATTRIB_ARRAY_ENABLED, ... VERTEX_ATTRIB_ARRAY_NORMALIZED, VERTEX_ATTRIB_ARRAY_INTEGER_EXT, or CURRENT_VERTEX_ATTRIB. ...

(split 3rd paragraph, p.259) ... The size, stride, type, normalized flag, and unconverted integer flag are set by the commands VertexAttribPointer and VertexAttribIPointerEXT. The normalized flag is always set to FALSE by by VertexAttribIPointerEXT. The unconverted integer flag is always set to FALSE by VertexAttribPointer and TRUE by VertexAttribIPointerEXT.

The query CURRENT_VERTEX_ATTRIB returns the current value for the generic attribute <index>. GetVertexAttribdv and GetVertexAttribfv read and return the current attribute values as floating-point values; GetVertexAttribiv reads them as floating-point values and converts them to integer values; GetVertexAttribIivEXT reads and returns them a signed integers; GetVertexAttribIuivEXT reads and returns them as unsigned integers. The results of the query are undefined if the current attribute values are read using one data type but specified using a different one. The error INVALID_OPERATION is generated if <index> is zero.

Additions to the AGL/GLX/WGL Specifications

None

GLX Protocol

TBD

Errors

None.

Dependencies on EXT_draw_instanced

If EXT_draw_instanced or a similar extension is not supported, references to the "vertex.instance" attribute binding and a primitive's instance number should be eliminated.

New State

(add to table 6.7, p. 268)

Get Value	Type	Get Command	Initial Value	Description	Sec.	Attribute
VERTEX_ATTRIB_ARRAY INTEGER_EXT	16+xB	GetVertexAttrib	FALSE	vertex attrib array has unconverted ints	2.8	vertex-array

New Implementation Dependent State

None.

Issues

(1) Should a new set of immediate-mode functions be provided for "real integer" attributes? If so, which ones should be provided?

RESOLVED: Yes, although an incomplete subset is provided. This extension provides vector and non-vector functions that accept 1-, 2-, 3-, and 4-component "int" and "uint" values. Additionally, we provide only 4-component vector versions of functions that accept "byte", "ubyte", "short", and "ushort" values. Note that the ARB_vertex_program extension provided a similar incomplete subset.

Since existing VertexAttrib functions include versions that take integer values and convert them to float, it was necessary to create a different way to specify integer values that are not converted. We created a new set of functions using capital letter "I" to denote "real integer" values.

This "I" approach is consistent with a similar choice made by ARB_vertex_program for the existing integer attribute functions. There are two methods of converting to floating point -- straight casts and normalization to [0,1] or [-1,+1]. The normalization version of the attribute functions use the capital letter "N" to denote normalization.

(2) For vertex arrays with "real integer" attributes, should we provide a new function to specify the array or re-use the existing one?

RESOLVED: Provide a new function, VertexAttribIPointerEXT. This function and VertexAttribPointer both set the same attribute state -- state set by VertexAttribPointer for a given <index> will be overwritten by VertexAttribIPointerEXT() and vice versa. There is one new piece of state per array (VERTEX_ATTRIB_ARRAY_INTEGER_EXT) which is set to TRUE for VertexAttribIPointerEXT() and FALSE by VertexAttribPointer. The use of a new function with capital "I" in the name is consistent with the choice made for immediate-mode integer attributes.

We considered reusing the existing VertexAttribPointer function by hijacking the <normalized> parameter, which specifies whether the provided arrays are converted to float by normalizing or a straight cast. It would have been possible to add a third setting to indicate unconverted integer values, but that has two problems: (a) it doesn't agree with the <normalized> flag being specified as a "boolean" (which only has two values), and (b) the enum value that would be used would be outside the range [0,255] and "boolean" may be represented using single-byte data types.

One other possibility would have been to create a new set of <type> values to indicate integer values that are never converted to floating point -- for example, GL_INTEGER_INT.

(3) Should we provide a whole new set of generic integer vertex attributes?

RESOLVED: No. This extension makes the existing generic vertex attributes "typeless", where they can store either integer or floating-point data. This avoids the need to introduce new hardware resources for integer vertex attributes or software overhead in juggling integer and floating-point generic attributes.

Vertex programs and any queries that access these attributes are responsible for ensuring that they are read using the same data type that they were specified using, and will get undefined results on type mismatches. Checking for such mismatches would be an excellent feature for an instrumented OpenGL driver, or other debugging tool.

(4) Should we provide integer forms of existing conventional attributes?

RESOLVED: No. We could have provided "integer" versions of Color, TexCoord, MultiTexCoord, and other functions, but it didn't seem useful. The use of generic attributes for such values is perfectly acceptable, and fixed-function vertex processing paths won't know what to do with integer values for position, color, normal, and so on.

(5) With integers throughout the pipeline, should we provide automatic identifiers that can be read to get a "vertex number"? If so, how should this functionality be provided?

RESOLVED: The "vertex.id" binding provides an integer "vertex number" for each vertex called the "vertex ID".

When using vertex arrays in vertex buffer objects (VBOs), the vertex ID is defined to be the index of the vertex in the array -- the value implicitly passed to ArrayElement() when DrawArrays() or DrawElements() is called. In practice, vertex arrays in buffer objects will be stored in memory that is directly accessible by the GPU. When functions such as DrawArrays() or DrawElements() are called, a set of vertex indices are passed to the GPU to identify the vertices to pull out of the buffer objects. These same indices can be easily passed to the vertex program.

Vertex IDs can be used by applications in a variety of ways, for example to compute or look up some property of the vertex based on its position in the data set.

Note: The EXT_texture_buffer_object extension can be used to bind a buffer object as a texture resource, which can the be used for lookups in a vertex program. If the amount of memory required for each vertex is very large or is variable, the existing vertex array model might not work very well. However, with TexBOs (texture buffer objects), the vertex program can be used to compute an offset into the buffer object holding the vertex data and fetch the data needed using texture lookups.

This approach blurs the line between texture and vertex pulling, and treats the "texture" in question as a simple array.

(6) Should vertex IDs be provided for vertices in immediate mode? Vertices in display lists? Vertex arrays compiled into a display list?

RESOLVED: No to all.

A different definition would be needed for immediate mode vertices, since the vertex attributes are not specified with an index. It would have been possible to implement some sort of counter where the vertex ID indicates that the vertex is the <N>th one since the previous Begin command.

Vertex arrays compiled into a display list are an even more complicated problem, since either the "array element" definition or the alternate "immediate mode" definition could be used. If the "array element" definition were used, it would additionally be necessary to compile the array element values into the display list. This would introduce additional overhead into the display list, and the storage space for the array element numbers would be wasted if no program using vertex ID were ever used.

While such functionality may be useful, it is left to a subsequent extension.

If such functionality is required, immediate mode VertexAttribI1i() calls can be used to specify the desired "vertex ID" values as integer generic attributes. In this case, the vertex program needs to refer to the specified generic attribute, and not "vertex.id".

(7) Should vertex identifiers be provided for non-VBO vertex arrays? For vertex arrays that are a mix of VBO and non-VBO arrays?

RESOLVED: Non-VBO arrays are generally not stored in memory directly accessible by the GPU; the data are instead copied from the application's memory as they are passed to the GPU. Additionally, the index ordering may not be preserved by the copy. For example, if a DrawElements() call passes vertices numbered 30, 20, 10, and 0 in order, the GPU might see vertex 30's data first, immediately followed by vertex 20's data, and so on.

It would be possible for the driver to provide per-vertex ID values to the GPU during the copy, but defining such functionality is left to a subsequent extension.

For vertices with a mix of VBO arrays and non-VBO arrays, the non-VBO arrays still have the same copy issues, so the automatic vertex ID is not provided.

If such functionality is required, a generic vertex attribute array can be set up using VertexAttribIPointerEXT(), holding integer values 0 through <maxSize>-1, where <maxSize> is the maximum vertex index used. For each vertex, the appropriate "vertex ID" value will be taken from this array. In this case, the vertex program needs to refer to the specified generic attribute, and not "vertex.id".

(8) Should vertex IDs be available to geometry programs, and if so, how?

RESOLVED: Yes, vertex IDs can be passed to geometry programs using the "result.id" binding in a vertex program. Note there is no requirement that the "result.id" written for a vertex must match the "vertex.id" originally provided.

Vertex IDs are not automatically provided to geometry programs; if a vertex program doesn't write to "result.id" or if fixed-function vertex processing is used, the vertex ID visible to the geometry program is undefined.

(9) For instanced arrays (EXT_draw_instanced), should a vertex program be able to read the instance number? If so, how?

RESOLVED: Yes, instance IDs are available to vertex programs using the "vertex.instance" attribute. The instance ID is available in the "x" component and should be read as an integer.

(10) Should instance IDs be available to geometry and fragment programs, and if so, how?

UNRESOLVED: No. If a geometry or fragment program needs the instance ID, the value read in the vertex program can be passed down using a generic integer vertex attribute.

It would be possible to provide a named output binding (e.g., "result.instance") that could be used to pass the instance ID to the next pipeline stage. Using such a binding would have no functional differences from using a generic attribute, except for a name.

In any event, instance IDs are not automatically available to geometry or fragment programs; they must be passed from earlier pipeline stages.

(11) This is an NV extension (NV_vertex_program4). Why do all the new functions and tokens have an "EXT" extension?

RESOLVED: These functions and tokens are shared between this extension and the comparable high-level GLSL programmability extension (EXT_gpu_shader4). Rather than provide a duplicate set of functions, we simply use the EXT version here.

Revision History

None

Name

    SGIS_generate_mipmap

Name Strings

    GL_SGIS_generate_mipmap

Version

```
    SGI Date: 1997/02/26 03:36:30  SGI Revision: 1.6
    $Id: //sw/main/docs/OpenGL/specs/GL_SGIS_generate_mipmap.txt#2 $
```

Number

    32

Dependencies

```
    EXT_texture is required
    EXT_texture3D affects the definition of this extension
    EXT_texture_object affects the definition of this extension
    SGIS_texture_lod affects the definition of this extension
```

Overview

    This extension defines a mechanism by which OpenGL can derive the
    entire set of mipmap arrays when provided with only the base level
    array. Automatic mipmap generation is particularly useful when
    texture images are being provided as a video stream.

Issues

    *   How are edges handled?

New Procedures and Functions

    None

New Tokens

```
    Accepted by the <pname> parameter of TexParameteri, TexParameterf,
    TexParameteriv, TexParameterfv, GetTexParameteriv, and GetTexParameterfv:

        GENERATE_MIPMAP_SGIS            0x8191

    Accepted by the <target> parameter of Hint, and by the <pname>
    parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

        GENERATE_MIPMAP_HINT_SGIS       0x8192
```

Additions to Chapter 2 of the 1.0 Specification (OpenGL Operation)

    None

Additions to Chapter 3 of the 1.0 Specification (Rasterization)

GL Specification Table 3.7 is updated as follows:

Name	Type	Legal Values
TEXTURE_WRAP_S	integer	CLAMP, REPEAT
TEXTURE_WRAP_T	integer	CLAMP, REPEAT
TEXTURE_WRAP_R_EXT	integer	CLAMP, REPEAT
TEXTURE_MIN_FILTER	integer	NEAREST, LINEAR, NEAREST_MIPMAP_NEAREST, NEAREST_MIPMAP_LINEAR, LINEAR_MIPMAP_NEAREST, LINEAR_MIPMAP_LINEAR, FILTER4_SGIS
TEXTURE_MAG_FILTER	integer	NEAREST, LINEAR, FILTER4_SGIS, LINEAR_DETAIL_SGIS, LINEAR_DETAIL_ALPHA_SGIS, LINEAR_DETAIL_COLOR_SGIS, LINEAR_SHARPEN_SGIS, LINEAR_SHARPEN_ALPHA_SGIS, LINEAR_SHARPEN_COLOR_SGIS
TEXTURE_BORDER_COLOR	4 floats	any 4 values in [0,1]
DETAIL_TEXTURE_LEVEL_SGIS	integer	any non-negative integer
DETAIL_TEXTURE_MODE_SGIS	integer	ADD, MODULATE
TEXTURE_MIN_LOD_SGIS	float	any value
TEXTURE_MAX_LOD_SGIS	float	any value
TEXTURE_BASE_LEVEL_SGIS	integer	any non-negative integer
TEXTURE_MAX_LEVEL_SGIS	integer	any non-negative integer
GENERATE_MIPMAP_SGIS	boolean	TRUE or FALSE

Table 3.7: Texture parameters and their values.

This extension introduces a side effect to the modification of the base level mipmap array. The side effect is enabled on a per-texture basis by calling TexParameteri, TexParameterf, TexParameteriv, or TexParameterfv with <target> set to TEXTURE_1D, TEXTURE_2D, or TEXTURE_3D_EXT, <pname> set to GENERATE_MIPMAP_SGIS, and <param> set to TRUE (or <params> pointing to TRUE). It is disabled using the same call, with <param> set to FALSE, or <params> pointing to FALSE. If SGIS_texture_lod is supported, the base level array is the array number TEXTURE_BASE_LEVEL_SGIS. Otherwise the base level array is array zero.

If GENERATE_MIPMAP_SGIS is enabled, the side effect occurs whenever any change is made to the interior or edge image values of the base level texture array. The side effect is computation of a complete set of mipmap arrays, all derived from the modified base level array. Array levels BASE+1 through BASE+p are replaced with derived arrays, regardless of their previous contents. All other texture arrays, including the base array, are left unchanged by this mipmap computation.

The internal formats and border widths of the derived mipmap arrays all match those of the base array, and the dimensions of the derived arrays follow the requirements described in the Mipmapping section of the GL Specification. The result is that the set of mipmap arrays is complete as defined by the GL Specification. The contents of the derived image arrays are computed by repeated, filtered reduction of the base level image array. This specification does not require any particular filter algorithm, though a simple 2x2 box filter is recommended as the default filter. Hint variable GENERATE_MIPMAP_HINT_SGIS can be changed from its default value of DONT_CARE to either FASTEST or NICEST, indicating to the implementation that either the fastest or highest quality filter operation is desired. These operations are not defined by this specification, however. The single hint value controls the filtering of all the textures, and is evaluated when the filtering operation takes place.

Automatic mipmap generation is available for texture targets TEXTURE_1D, TEXTURE_2D, and TEXTURE_3D_EXT only.

Additions to Chapter 4 of the 1.0 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.0 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.0 Specification (State and State Requests)

None

Additions to the GLX Specification

None

Dependencies on EXT_texture

EXT_texture is required.

Dependencies on EXT_texture3D

If EXT_texture3D is not supported, references to 3D texture mapping and to TEXTURE_3D_EXT in this document are invalid and should be ignored.

Dependencies on EXT_texture_object

If EXT_texture_object is implemented, the state value named

    GENERATE_MIPMAP_SGIS

is added to the state vector of each texture object. When an attribute set that includes texture information is popped, the bindings and enables are first restored to their pushed values, then the bound textures have their GENERATE_MIPMAP_SGIS parameters restored to their pushed values.

Dependencies on SGIS_texture_lod

If SGIS_texture_lod is not supported, the base array level is always level zero. References in this document to TEXTURE_BASE_LEVEL_SGIS should be ignored.

Errors

None

New State

Get Value	Get Command	Type	Value	Initial Attrib
GENERATE_MIPMAP_SGIS	GetTexParameteriv	B	FALSE	texture
GENERATE_MIPMAP_HINT_SGIS	GetIntegerv	Z3	DONT_CARE	hint

New Implementation Dependent State

None

Name

    SGIS_texture_lod

Name Strings

    GL_SGIS_texture_lod

Version

    $Date: 1997/05/30 01:34:44 $ $Revision: 1.8 $

Number

    24

Dependencies

    EXT_texture is required
    EXT_texture3D affects the definition of this extension
    EXT_texture_object affects the definition of this extension
    SGI_detail_texture affects the definition of this extension
    SGI_sharpen_texture affects the definition of this extension

Overview

This extension imposes two constraints related to the texture level of detail parameter LOD, which is represented by the Greek character lambda in the GL Specification. One constraint clamps LOD to a specified floating point range. The other limits the selection of mipmap image arrays to a subset of the arrays that would otherwise be considered.

Together these constraints allow a large texture to be loaded and used initially at low resolution, and to have its resolution raised gradually as more resolution is desired or available. Image array specification is necessarily integral, rather than continuous. By providing separate, continuous clamping of the LOD parameter, it is possible to avoid "popping" artifacts when higher resolution images are provided.

Note: because the shape of the mipmap array is always determined by the dimensions of the level 0 array, this array must be loaded for mipmapping to be active. If the level 0 array is specified with a null image pointer, however, no actual data transfer will take place. And a sufficiently tuned implementation might not even allocate space for a level 0 array so specified until true image data were presented.

Issues

* Should detail and sharpen texture operate when the level 0 image is not being used?

  A: Sharpen yes, detail no.

* Should the shape of the mipmap array be determined by the dimensions of the level 0 array, regardless of the base level?

```
        A: Yes, this is the better solution.  Driving everything from
           the base level breaks the proxy query process, and allows
           mipmap arrays to be placed arbitrarily.  The issues of
           requiring a level 0 array are partially overcome by the use
           of null-point loads, which avoid data transfer and,
           potentially, data storage allocation.

    *   With the arithmetic as it is, a linear filter might access an
        array past the limit specified by MAX_LEVEL or p.  But the
        results of this access are not significant, because the blend
        will weight them as zero.
```

New Procedures and Functions

```
    None
```

New Tokens

```
    Accepted by the <pname> parameter of TexParameteri, TexParameterf,
    TexParameteriv, TexParameterfv, GetTexParameteriv, and GetTexParameterfv:

        TEXTURE_MIN_LOD_SGIS            0x813A
        TEXTURE_MAX_LOD_SGIS            0x813B
        TEXTURE_BASE_LEVEL_SGIS         0x813C
        TEXTURE_MAX_LEVEL_SGIS          0x813D
```

Additions to Chapter 2 of the 1.0 Specification (OpenGL Operation)

```
    None
```

Additions to Chapter 3 of the 1.0 Specification (Rasterization)

GL Specification Table 3.7 is updated as follows:

Name	Type	Legal Values
TEXTURE_WRAP_S	integer	CLAMP, REPEAT
TEXTURE_WRAP_T	integer	CLAMP, REPEAT
TEXTURE_WRAP_R_EXT	integer	CLAMP, REPEAT
TEXTURE_MIN_FILTER	integer	NEAREST, LINEAR, NEAREST_MIPMAP_NEAREST, NEAREST_MIPMAP_LINEAR, LINEAR_MIPMAP_NEAREST, LINEAR_MIPMAP_LINEAR, FILTER4_SGIS
TEXTURE_MAG_FILTER	integer	NEAREST, LINEAR, FILTER4_SGIS, LINEAR_DETAIL_SGIS, LINEAR_DETAIL_ALPHA_SGIS, LINEAR_DETAIL_COLOR_SGIS, LINEAR_SHARPEN_SGIS, LINEAR_SHARPEN_ALPHA_SGIS, LINEAR_SHARPEN_COLOR_SGIS
TEXTURE_BORDER_COLOR	4 floats	any 4 values in [0,1]
DETAIL_TEXTURE_LEVEL_SGIS	integer	any non-negative integer
DETAIL_TEXTURE_MODE_SGIS	integer	ADD, MODULATE
TEXTURE_MIN_LOD_SGIS	float	any value
TEXTURE_MAX_LOD_SGIS	float	any value
TEXTURE_BASE_LEVEL_SGIS	integer	any non-negative integer
TEXTURE_MAX_LEVEL_SGIS	integer	any non-negative integer

Table 3.7: Texture parameters and their values.

Base Array

Although it is not explicitly stated, it is the clear intention of the OpenGL specification that texture minification filters NEAREST and LINEAR, and all texture magnification filters, be applied to image array zero. This extension introduces a parameter, BASE_LEVEL, that explicitly specifies which array level is used for these filter operations. Base level is specified for a specific texture by calling TexParameteri, TexParameterf, TexParameteriv, or TexParameterfv with <target> set to TEXTURE_1D, TEXTURE_2D, or TEXTURE_3D_EXT, <pname> set to TEXTURE_BASE_LEVEL_SGIS, and <param> set to (or <params> pointing to) the desired value. The error INVALID_VALUE is generated if the specified BASE_LEVEL is negative.

Level of Detail Clamping

The level of detail parameter LOD is defined in the first paragraph of Section 3.8.1 (Texture Minification) of the GL Specification, where it is represented by the Greek character lambda. This extension redefines the definition of LOD as follows:

```
    LOD'(x,y) = log_base_2 (Q(x,y))


          /  MAX_LOD       LOD' > MAX_LOD
    LOD = (   LOD'          LOD' >= MIN_LOD and LOD' <= MAX_LOD
          \  MIN_LOD       LOD' < MIN_LOD
           \ undefined     MIN_LOD > MAX_LOD
```

The variable Q in this definition represents the Greek character rho, as it is used in the OpenGL Specification. (Recall that Q is computed based on the dimensions of the BASE_LEVEL image array.) MIN_LOD is the value of the per-texture variable TEXTURE_MIN_LOD_SGIS, and MAX_LOD is the value of the per-texture variable TEXTURE_MAX_LOD_SGIS.

Initially TEXTURE_MIN_LOD_SGIS and TEXTURE_MAX_LOD_SGIS are -1000 and 1000 respectively, so they do not interfere with the normal operation of texture mapping. These values are respecified for a specific texture by calling TexParameteri, TexParemeterf, TexParameteriv, or TexParameterfv with <target> set to TEXTURE_1D, TEXTURE_2D, or TEXTURE_3D_EXT, <pname> set to TEXTURE_MIN_LOD_SGIS or TEXTURE_MAX_LOD_SGIS, and <param> set to (or <params> pointing to) the new value. It is not an error to specify a maximum LOD value that is less than the minimum LOD value, but the resulting LOD values are not defined.

LOD is clamped to the specified range prior to any use. Specifically, the mipmap image array selection described in the Mipmapping Subsection of the GL Specification is based on the clamped LOD value. Also, the determination of whether the minification or magnification filter is used is based on the clamped LOD.

Mipmap Completeness

The GL Specification describes a "complete" set of mipmap image arrays as array levels 0 through p, where p is a well defined function of the dimensions of the level 0 image. This extension modifies the notion of completeness: instead of requiring that all arrays 0 through p meet the requirements, only arrays 0 and arrays BASE_LEVEL through MAX_LEVEL (or p, whichever is smaller) must meet these requirements. The specification of BASE_LEVEL was described above. MAX_LEVEL is specified by calling TexParameteri, TexParemeterf, TexParameteriv, or TexParameterfv with <target> set to TEXTURE_1D, TEXTURE_2D, or TEXTURE_3D_EXT, <pname> set to TEXTURE_MAX_LEVEL_SGIS, and <param> set to (or <params> pointing to) the desired value. The error INVALID_VALUE is generated if the specified MAX_LEVEL is negative. If MAX_LEVEL is smaller than BASE_LEVEL, or if BASE_LEVEL is greater than p, the set of arrays is incomplete.

Array Selection

Magnification and non-mipmapped minification are always performed using only the BASE_LEVEL image array. If the minification filter is one that requires mipmapping, one or two array levels are selected using the equations in the table below, and the LOD value is clamped to a maximum value that insures that no array beyond the limits specified by MAX_LEVEL and p is accessed.

Minification Filter	Maximum LOD	Array level(s)
NEAREST_MIPMAP_NEAREST	M + 0.4999	floor(B + 0.5)
LINEAR_MIPMAP_NEAREST	M + 0.4999	floor(B + 0.5)
NEAREST_MIPMAP_LINEAR	M	floor(B), floor(B)+1
LINEAR_MIPMAP_LINEAR	M	floor(B), floor(B)+1

where:

```
    M = min(MAX_LEVEL,p) - BASE_LEVEL
    B = BASE_LEVEL + LOD
```

For NEAREST_MIPMAP_NEAREST and LINEAR_MIPMAP_NEAREST the specified image array is filtered according to the rules for NEAREST or LINEAR respectively. For NEAREST_MIPMAP_LINEAR and LINEAR_MIPMAP_LINEAR both selected arrays are filtered according to the rules for NEAREST or LINEAR, respectively. The resulting values are then blended as described in the Mipmapping section of the OpenGL specification.

Additional Filters

Sharpen filters (described in SGIS_sharpen_texture) operate on array levels BASE_LEVEL and BASE_LEVEL+1. If the minimum of MAX_LEVEL and p is not greater than BASE_LEVEL, then sharpen texture reverts to a LINEAR magnification filter. Detail filters (described in SGIS_detail_texture) operate only when BASE_LEVEL is zero.

Texture Capacity

In Section 3.8 the OpenGL specification states:

"In order to allow the client to meaningfully query the maximum image array sizes that are supported, an implementation must not allow an image array of level one or greater to be created if a `complete' set of image arrays consistent with the requested array could not be supported."

Given this extension's redefinition of completeness, the above paragraph should be rewritten to indicate that all levels of the `complete' set of arrays must be supportable. E.g.

"In order to allow the client to meaningfully query the maximum image array sizes that are supported, an implementation must not allow an image array of level one or greater to be created if a `complete' set of image arrays (all levels 0 through p) consistent with the requested array could not be supported."

Additions to Chapter 4 of the 1.0 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.0 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.0 Specification (State and State Requests)

None

Additions to the GLX Specification

None

Dependencies on EXT_texture

EXT_texture is required.

Dependencies on EXT_texture3D

If EXT_texture3D is not supported, references to 3D texture mapping and to TEXTURE_3D_EXT in this document are invalid and should be ignored.

Dependencies on EXT_texture_object

If EXT_texture_object is implemented, the state values named

```
TEXTURE_MIN_LOD_SGIS
TEXTURE_MAX_LOD_SGIS
TEXTURE_BASE_LEVEL_SGIS
TEXTURE_MAX_LEVEL_SGIS
```

are added to the state vector of each texture object. When an attribute set that includes texture information is popped, the bindings and enables are first restored to their pushed values, then the bound textures have their LOD and LEVEL parameters restored to their pushed values.

Dependencies on SGIS_detail_texture

If SGIS_detail_texture is not supported, references to detail texture mapping in this document are invalid and should be ignored.

Dependencies on SGIS_sharpen_texture

If SGIS_sharpen_texture is not supported, references to sharpen texture mapping in this document are invalid and should be ignored.

Errors

INVALID_VALUE is generated if an attempt is made to set TEXTURE_BASE_LEVEL_SGIS or TEXTURE_MAX_LEVEL_SGIS to a negative value.

New State

Get Value	Get Command	Type	Initial Value	Attrib
TEXTURE_MIN_LOD_SGIS	GetTexParameterfv	n x R	-1000	texture
TEXTURE_MAX_LOD_SGIS	GetTexParameterfv	n x R	1000	texture
TEXTURE_BASE_LEVEL_SGIS	GetTexParameteriv	n x R	0	texture
TEXTURE_MAX_LEVEL_SGIS	GetTexParameteriv	n x R	1000	texture

New Implementation Dependent State

None

Name

```
    SGIX_depth_texture
```

Name Strings

```
    GL_SGIX_depth_texture
```

Version

```
    $Date: 1997/02/26 03:36:29 $ $Revision: 1.5 $
    $Id: //sw/main/docs/OpenGL/specs/GL_SGIX_depth_texture.txt#3 $
```

Number

```
    63
```

Dependencies

```
    EXT_texture is required
    EXT_subtexture affects the definition of this extension
    EXT_copy_texture affects the definition of this extension
```

Overview

```
    This extension defines a new depth texture format.  An important
    application of depth texture images is shadow casting, but separating
    this from the shadow extension allows for the potential use of
    depth textures in other applications such as image-based rendering
    or displacement mapping.  This extension does not define new
    depth-texture environment functions, such as filtering or applying
    the depth values computed from a texture, but leaves this to other
    extensions, such as the shadow extension.
```

New Procedures and Functions

```
    None
```

New Tokens

```
    Accepted by the <components> parameters of TexImage1D and
    TexImage2D, and by the <internalformat> parameters of TexImage3DEXT,
    CopyTexImage1DEXT, and CopyTexImage2DEXT:

    DEPTH_COMPONENT16_SGIX              0x81A5
    DEPTH_COMPONENT24_SGIX              0x81A6
    DEPTH_COMPONENT32_SGIX              0x81A7
```

Additions to Chapter 2 of the 1.0 Specification (OpenGL Operation)

```
    None
```

Additions to Chapter 3 of the 1.0 Specification (Rasterization)

```
    XXX - lots

    Notes:
```

* Defines DEPTH_COMPONENT as a new base internal format for textures. Defines 16, 24, and 32 bit specific internal formats for texture. Just as for the specific color internal formats, an implementation can choose whether to implement them or not.

* Texture commands that accept images from memory now allow the internal format to be DEPTH_COMPONENT or DEPTH_COMPONENT*_SGIX when the format of the image data is DEPTH_COMPONENT. Depth, not color pixel transfer operations are applied to depth images.

* Texture commands that accept images from the framebuffer now take their data from the depth buffer when the internal format is DEPTH_COMPONENT or DEPTH_COMPONENT*_SGIX, or when no internal format is specified, and the internal format of the target texture is DEPTH_COMPONENT or DEPTH_COMPONENT*_SGIX.

Additions to Chapter 4 of the 1.0 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.0 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.0 Specification (State and State Requests)

None

Additions to the GLX Specification

None

Dependencies on EXT_texture

EXT_texture is required.

Dependencies on EXT_texture3D

EXT_texture3D is not required, but if it is not supported, the implementation must compute the R texture coordinate as if it were. If EXT_texture3D is not supported, references to TexImage3DEXT and TexSubImage3DEXT in this document are invalid and should be ignored.

Dependencies on EXT_subtexture

If EXT_subtexture is not supported, references to TexSubImage1DEXT, TexSubImage2DEXT, and TexSubImage3DEXT in this document are invalid and should be ignored. If EXT_subtexture is supported, the operations of these three commands are affected by this extension.

Dependencies on EXT_copy_texture

If EXT_copy_texture is not supported, references to CopyTexImage1DEXT and CopyTexImage2DEXT in this document are invalid and should be ignored. If EXT_copy_texture is supported, the operations of these

    two commands, and of CopyTexSubImage1DEXT, CopyTexSubImage2DEXT,
    and CopyTexSubImage3DEXT are affected by this extension.

Errors

    INVALID_OPERATION is generated if TexImage1D or TexImage2D parameter
    <format> is DEPTH_COMPONENT and parameter <components> is not
    DEPTH_COMPONENT, DEPTH_COMPONENT16_SGI, DEPTH_COMPONENT24_SGI,
    or DEPTH_COMPONENT32_SGI.

    INVALID_OPERATION is generated if TexImage3DEXT parameter
    <format> is DEPTH_COMPONENT and parameter <internalformat> is not
    DEPTH_COMPONENT, DEPTH_COMPONENT16_SGI, DEPTH_COMPONENT24_SGI,
    or DEPTH_COMPONENT32_SGI.

    INVALID_OPERATION is generated if CopyTexImage1DEXT
    or CopyTexImage2DEXT parameter <internalformat> is
    DEPTH_COMPONENT, DEPTH_COMPONENT16_SGI, DEPTH_COMPONENT24_SGI,
    or DEPTH_COMPONENT32_SGI, and there is no depth buffer.

New State

    None

New Implementation Dependent State

    None

Name

    SGIX_shadow

Name Strings

    GL_SGIX_shadow

Version

```
$Date: 1997/08/27 19:54:45 $ $Revision: 1.15 $
$Id: //sw/main/docs/OpenGL/specs/GL_SGIX_shadow.txt#4 $
```

Number

    34

Dependencies

    None.

Overview

    This extension defines two new operations to be performed
    on texture values before they are passed on to the filtering
    subsystem. These operations perform either a <= or >= test
    on the value from texture memory and the iterated R value,
    and return 1.0 or 0.0 if the test passes or fails, respectively.

New Procedures and Functions

    None

New Tokens

```
Accepted by the <pname> parameter of TexParameterf, TexParameteri,
TexParameterfv, TexParameteriv, GetTexParameterfv, and
GetTexParameteriv, with the <pname> parameter of TRUE or FALSE:

TEXTURE_COMPARE_SGIX

Accepted by the <pname> parameter of TexParameterf, TexParameteri,
TexParameterfv, TexParameteriv, GetTexParameterfv, and
GetTexParameteriv:

TEXTURE_COMPARE_OPERATOR_SGIX

Accepted by the <param> parameter of TexParameterf and
TexParameteri, and by the <params> parameter of TexParameterfv
and TexParameteriv, when their <pname> parameter is
TEXTURE_COMPARE_OPERATOR_SGIX:

TEXTURE_LEQUAL_R_SGIX
TEXTURE_GEQUAL_R_SGIX
```

Additions to Chapter 2 of the 1.0 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.0 Specification (Rasterization)

XXX - lots

GL Specification Table 3.8 is updated as follows:

Name	Type	Legal Values
TEXTURE_WRAP_S	integer	CLAMP, REPEAT
TEXTURE_WRAP_T	integer	CLAMP, REPEAT
TEXTURE_WRAP_R_EXT	integer	CLAMP, REPEAT
TEXTURE_MIN_FILTER	integer	NEAREST, LINEAR, NEAREST_MIPMAP_NEAREST, NEAREST_MIPMAP_LINEAR, LINEAR_MIPMAP_NEAREST, LINEAR_MIPMAP_LINEAR, FILTER4_SGIS, LINEAR_CLIPMAP_LINEAR_SGIX
TEXTURE_MAG_FILTER	integer	NEAREST, LINEAR, FILTER4_SGIS, LINEAR_DETAIL_SGIS, LINEAR_DETAIL_ALPHA_SGIS, LINEAR_DETAIL_COLOR_SGIS, LINEAR_SHARPEN_SGIS, LINEAR_SHARPEN_ALPHA_SGIS, LINEAR_SHARPEN_COLOR_SGIS,
TEXTURE_BORDER_COLOR	4 floats	any 4 values in [0,1]
DETAIL_TEXTURE_LEVEL_SGIS	integer	any non-negative integer
DETAIL_TEXTURE_MODE_SGIS	integer	ADD, MODULATE
TEXTURE_MIN_LOD_SGIS	float	any value
TEXTURE_MAX_LOD_SGIS	float	any value
TEXTURE_BASE_LEVEL_SGIS	integer	any non-negative integer
TEXTURE_MAX_LEVEL_SGIS	integer	any non-negative integer
GENERATE_MIPMAP_SGIS	boolean	TRUE or FALSE
TEXTURE_CLIPMAP_OFFSET_SGIX	2 floats	any 2 values
TEXTURE_COMPARE_SGIX	boolean	TRUE or FALSE
TEXTURE_COMPARE_OPERATOR_SGIX	integer	TEXTURE_LEQUAL_R_SGIX, TEXTURE_GEQUAL_R_SGIX

Table 3.8: Texture parameters and their values.

Notes:

* Two new texture operators are defined which alter the sampled texture values before they are filtered. These operators are defined only for textures with internal format DEPTH_COMPONENT or DEPTH_COMPONENTS*_SGI.

* The new operators compare the sample texel value to the value of the third texture coordinate, R. The texture components are treated as though they range from 0.0 through 1.0. The value of the test is zero if the test fails, and one if it passes.

* The test for operator TEXTURE_LEQUAL_R_SGIX passes if the texel value is less than or equal to R. The test for operator

TEXTURE_GEQUAL_R_SGIX passes if the texel value is greater than or equal to R.

* The modified texels (with value 0.0 or 1.0 depending on the test result) are treated as if the texture internal format were LUMINANCE.

Additions to Chapter 4 of the 1.0 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.0 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.0 Specification (State and State Requests)

The texture compare operator is queried by calling GetTexParameteriv and GetTexParameterfv with <pname> set to TEXTURE_COMPARE_OPERATOR_SGIX. Texture compare enable/disable state is queried by calling GetTexParameteriv or GetTexParameterif with <pname> TEXTURE_COMPARE_SGIX.

Additions to the GLX Specification

None

Errors

INVALID_OPERATION is generated if TexParameter[if] parameter <pname> is TEXTURE_COMPARE_OPERATOR_SGIX and parameter <param> is not TEXTURE_LEQUAL_R_SGIX,or TEXTURE_GEQUAL_R_SGIX.

New State

Get Value	Get Command	Type	Initial Value	Attribute
TEXTURE_COMPARE_SGIX	GetTexParameter[if]v	B	False	texture
TEXTURE_COMPARE_OPERATOR_SGIX	GetTexParameter[if]v	Z_2	TEXTURE_LEQUAL_R_SGIX	texture

New Implementation Dependent State

None

NVIDIA Implementation Details

The specification is unclear if the R texture coordinate is clamped to the range [0,1]. NVIDIA hardware supporting this extension does clamp the R texture coordinate to the range [0,1] on a per-fragment basis.

The behavior of the NV_register_combiners SIGNED_NEGATE_NV mapping mode is undefined when used to map the initial value of a texture register corresponding to an enabled texture with a base internal format of GL_DEPTH_COMPONENT and a true TEXTURE_COMPARE_SGIX mode when multiple enabled textures have different values for TEXTURE_COMPARE_OPERATOR_SGIX. . Values subsequently assigned

```
        to such registers and then mapped with SIGNED_NEGATIE_NV operate
        as expected.
```

Name

    SUN_slice_accum

Name Strings

    GL_SUN_slice_accum

Contact

    Jack Middleton, Sun (Jack.Middleton 'at' sun.com)

Status

    Shipping (version 1.3)

Version

 $Date: 02/03/13 15:15:35 $Revision: 1.3 $

Number

    258

Dependencies

    The extension is written against the OpenGL 1.3 Specification.

Overview

  This extension defines a new accumulation operation which enables the
  accumulation buffer to be used for alpha compositing. This enables
  higher precision alpha blending than what can be accomplished using
  the blend operation.

IP Status

  There are no known IP issues.

Issues

  None

New Procedures and Functions

  None

New Tokens

```
   Accepted by the <op> parameter of Accum,

   SLICE_ACCUM_SUN                 0x85CC
```

Additions to Chapter 2 of the 1.3 GL Specification (OpenGL Operation)

    None

Additions to Chapter 3 of the 1.3 GL Specification (Rasterization)

None

Additions to Chapter 4 of the 1.3 GL Specification (Per-Fragment Operations and the Framebuffer)

Section 4.2.4 The Accumulation Buffer:

The possible operations are ACCUM, LOAD, RETURN, MULT, ADD and SLICE_ACCUM_SUN.

The SLICE_ACCUM_SUN operation has the same effect as ACCUM except that the accumulation buffer color value is computed:

```
    AccumRGB = (FrameBuffAlpha * FrameBuffRGB) + ((1 - FrameBuffAlpha) * AccumRGB)
```

Additions to Chapter 5 of the 1.3 GL Specification (Special Functions)

None

Additions to Chapter 6 of the 1.3 GL Specification (State and State Requests)

None

Additions to the GLX / WGL / AGL Specifications

None

GLX Protocol

None

Errors

None

New State

None

Name

    EXT_texture_from_pixmap

Name Strings

    GLX_EXT_texture_from_pixmap

Status

    Complete

Version

    16 (12 Sep 2006)

Number

    344

Dependencies

    OpenGL 1.1 is required.
    GLX 1.3 is required.
    GL_EXT_framebuffer_object affects the definition of this extension.
    GL_ARB_texture_rectangle affects the definition of this extension.
    GL_ARB_texture_non_power_of_two affects the definition of this extension.
    GL_SGIS_generate_mipmap affects the definition of this extension.

Overview

    This extension allows a color buffer to be used for both rendering and
    texturing.

    Only color buffers of pixmaps can be used for texturing by this extension
    but other types of drawables can be supported by future extensions layered
    on top of this extension.

    The functionality of this extension is similar to WGL_ARB_render_texture.
    However, the purpose of this extension is not to provide
    "render to texture" like functionality but rather the ability to bind
    an existing X drawable to a texture. Though, there is nothing that
    prohibits it from being used for "render to texture".

    -   Windows are problematic as they can change size and therefore are not
        supported by this extension.

    -   Only a color buffer of a GLX pixmap created using an FBConfig with
        attribute GLX_BIND_TO_TEXTURE_RGB_EXT or GLX_BIND_TO_TEXTURE_RGBA_EXT
        set to TRUE can be bound as a texture.

    -   The texture internal format is determined when the color buffer
        is associated with the texture, guaranteeing that the color
        buffer format is equivalent to the texture internal format.

    -   A client can create a complete set of mipmap images if
        EXT_framebuffer_object is supported.

IP Status

    There are no known IP issues.

Issues

    1. What should this extension be called?

    Even though it is very similar to WGL_ARB_render_texture that name is
    not appropriate as the intention of this extension is not
    "render to texture" like functionality.

    EXT_texture_from_pixmap seams most appropriate. Layering of future
    extensions on top of this extension for using other type of drawables
    as textures follows the same conventions as vertex/pixel buffer objects
    and vertex/fragment programs.

    2. Should we allow applications to render to different mipmap levels and
    cube map faces?

    In order to discourage the use of this extension as a render to texture
    mechanism, cube maps and rendering directly to mip-map levels > 0 will
    not be supported. A new FBConfig attribute is introduced that specifies
    whether or not drawables created with that config will support multiple
    mipmap levels when bound to a texture. The other mipmap levels can be
    filled in by the EXT_framebuffer_object GenerateMipmapEXT function.

    Specifying which level of a pixmap was being rendered to on a per-drawable
    basis, as was done in the WGL_ARB_render_texture extension, also
    introduces concurrency issues. The state of the drawable when it was
    being rendered two by two separate threads of execution and both were
    changing the mipmap level was difficult to define.

    It is also desireable to keep this extension as simple as possible.
    Adding functionality that complicates the implementation and that is not
    directly relevenat to the goal of exposing a mechanism for texturing from
    arbitrary X pixmaps is not productive. If the ability to render directly
    to all levels of a texture is needed, EXT_framebuffer_object is the
    extension that should be used.

    3. Should 1D textures be supported?

    X servers layered on top of an OpenGL implementation might not be able
    to support this. A new FBConfig attribute is introduced specifying
    which texture targets a drawable created with the given FBConfig can
    be bound to.

    4. What should the default value for GLX_TEXTURE_TARGET_EXT be? Should
    users be required to set this value if GLX_TEXTURE_FORMAT_EXT is not
    GLX_TEXTURE_FORMAT_NONE_EXT?

    The implementation is capable of choosing a reasonable default, we simply
    need to specify the correct way to do so. We can base the ordering on

```
    the properties of the pixmap and the texturing capabilities of the
    pixmap's FBConfig and the implementation.

    The order is:

    - If GL_ARB_texture_non_power_of_two is supported GL_TEXTURE_2D will
      be used for all pixmap sizes.

    - If only GL_ARB_texture_rectangle is supported GL_TEXTURE_2D will
      be used for all power of two pixmap sizes and GL_TEXTURE_RECTANGLE_ARB
      will be used for all non power of two pixmap sizes.


    5. Should users be required to re-bind the drawable to a texture after
    the drawable has been rendered to?

    It is difficult to define what the contents of the texture would be if
    we don't require this.  Also, requiring this would allow implementations
    to perform an implicit copy at this point if they could not support
    texturing directly out of renderable memory.

    The problem with defining the contents of the texture after rendering
    has occured to the associated drawable is that there is no way to
    synchronize the use of the buffer as a source and as a destination.
    Direct OpenGL rendering is not necessarily done in the same command
    stream as X rendering.  At the time the pixmap is used as the source
    for a texturing operation, it could be in a state halfway through a
    copyarea operation in which half of it is say, white, and half is the
    result of the copyarea operation.  How is this defined?  Worse, some
    other OpenGL application could be halfway through a frame of rendering
    when the composite manager sources from it.  The buffer might just
    contain the results of a "glClear" operation at that point.

    To gurantee tear-free rendering, a composite manager would run as follows:

    -receive request for compositing:
    XGrabServer()
    glXWaitX() or XSync()
    glXBindTexImageEXT()

    <Do rendering/compositing>

    glXReleaseTexImageEXT()
    XUngrabServer()

    Apps that don't synchronize like this would get what's available,
    and that may or may not be what they expect.


    6. What is the result of calling GenerateMipmapEXT on a drawable that
    was not created with mipmap levels?

    The results are undefined.


    7. Rendering done by the window system may be y-inverted compared
    to the standard OpenGL texture representation.  More specifically:
```

the X Window system uses a coordinate system where the origin is in the upper left; however, the GL uses a coordinate system where the origin is in the lower left. Should we define the contents of the texture as the y-inverted contents of the drawable?

X implementations may represent their drawables differently internally, so y-inversion should be exposed as an FBConfig attribute. Applications will need to query this attribute and adjust their rendering appropriately.

If a drawables is y-inverted and is bound to a texture, the contents of the texture will be y-inverted with respect to the standard GL memory layout. This means the contents of a pixmap of size (width, height) at pixmap coordinate (x, y) will be at location (x, height-y-1) in the texture. Applications will need to adjust their texture coordinates accordingly to avoid drawing the texture contents upside down.

8. Why wasn't this extension based on FBO instead of ARB_render_texture? Isn't the render_texture extension deprecated?

At first glance, FBO may seem like the perfect framework to base a spec for texturing from pixmap surfaces on. It replaced the WGL_ARB_render_texture specification, which provided a mechanism to texture from pbuffer surfaces. However, this train of thought is another side affect of the unfortunate naming of the WGL_ARB_render_texture specification. FBO and the orginal render_texture specification were two different solutions to the problem of how to render to and texture from the same surface. WGL_ARB_render_texture provided a method to bind a texture to a drawable surface, as this extension does. FBO provides the opposite solution, allowing rendering to arbitrary surfaces including textures. In the case of FBO, the application doing the rendering knows that it needs to render to an alternate surface. In our usage case, the application doing the rendering is arbitrary, and has no knowledge that another application wants to use the surface it is rendering to as a texture. The only application able to name the surface is the one texturing from it. Therefore, it makes sense to provide a mechanism for binding a texture to an arbitrary surface in general, and a pixmap in this particular case.

9. Why not allow binding directly to an X pixmap without creating an intermediate GLX pixmap?

Architecturally, GLX has moved away from operating directly on X drawables. This allows GLX specific attributes to be associated with the GLX drawables. In this case, it is important to associate an FBConfig with the drawable. The FBConfig contains attributes specifying the internal format the GL will use when utilizing the drawable's framebuffer as a texture.

New Procedures and Functions

```
    void glXBindTexImageEXT (Display     *display,
                             GLXDrawable drawable,
                             int         buffer,
                             const int   *attrib_list)
    void glXReleaseTexImageEXT (Display     *display,
                                GLXDrawable drawable,
                                int         buffer)
```

New Tokens

```
    Accepted by the <Attribute> parameter of glXGetFBConfigAttrib and
    the <attrib_list> parameter of glXChooseFBConfig:

        GLX_BIND_TO_TEXTURE_RGB_EXT        0x20D0
        GLX_BIND_TO_TEXTURE_RGBA_EXT       0x20D1
        GLX_BIND_TO_MIPMAP_TEXTURE_EXT     0x20D2
        GLX_BIND_TO_TEXTURE_TARGETS_EXT    0x20D3
        GLX_Y_INVERTED_EXT                 0x20D4

    Accepted as an attribute in the <attrib_list> parameter of
    glXCreatePixmap, and by the <attribute> parameter of glXQueryDrawable:

        GLX_TEXTURE_FORMAT_EXT             0x20D5
        GLX_TEXTURE_TARGET_EXT             0x20D6
        GLX_MIPMAP_TEXTURE_EXT             0x20D7

    Accepted as a value in the <attrib_list> parameter of glXCreatePixmap
    and returned in the <value> parameter of glXQueryDrawable when
    <attribute> is GLX_TEXTURE_FORMAT_EXT:

        GLX_TEXTURE_FORMAT_NONE_EXT        0x20D8
        GLX_TEXTURE_FORMAT_RGB_EXT         0x20D9
        GLX_TEXTURE_FORMAT_RGBA_EXT        0x20DA

    Accepted as bits in the GLX_BIND_TO_TEXTURE_TARGETS_EXT variable:

        GLX_TEXTURE_1D_BIT_EXT             0x00000001
        GLX_TEXTURE_2D_BIT_EXT             0x00000002
        GLX_TEXTURE_RECTANGLE_BIT_EXT      0x00000004

    Accepted as a value in the <attrib_list> parameter of glXCreatePixmap
    and returned in the <value> parameter of glXQueryDrawable when
    <attribute> is GLX_TEXTURE_TARGET_EXT:

        GLX_TEXTURE_1D_EXT                 0x20DB
        GLX_TEXTURE_2D_EXT                 0x20DC
        GLX_TEXTURE_RECTANGLE_EXT          0x20DD

    Accepted by the <Buffer> parameter of glXBindTexImageEXT and
    glXReleaseTexImageEXT:

        GLX_FRONT_LEFT_EXT                 0x20DE
        GLX_FRONT_RIGHT_EXT                0x20DF
        GLX_BACK_LEFT_EXT                  0x20E0
        GLX_BACK_RIGHT_EXT                 0x20E1
        GLX_FRONT_EXT                      GLX_FRONT_LEFT_EXT
        GLX_BACK_EXT                       GLX_BACK_LEFT_EXT
        GLX_AUX0_EXT                       0x20E2
        GLX_AUX1_EXT                       0x20E3
        GLX_AUX2_EXT                       0x20E4
        GLX_AUX3_EXT                       0x20E5
        GLX_AUX4_EXT                       0x20E6
        GLX_AUX5_EXT                       0x20E7
        GLX_AUX6_EXT                       0x20E8
        GLX_AUX7_EXT                       0x20E9
        GLX_AUX8_EXT                       0x20EA
        GLX_AUX9_EXT                       0x20EB
```

GLX Protocol

Two new GLX protocol commands are added.

```
        BindTexImageEXT
            1           CARD8                   opcode (X assigned)
            1           16                      GLX opcode (glXVendorPrivate)
            2           6+n                     request length
            4           1330                    vendor specific opcode
            4           CARD32                  context tag
            4           GLX_DRAWABLE            drawable
            4           INT32                   buffer
            4           CARD32                  num_attributes
            4*n         LISTofATTRIBUTE_PAIR    attribute, value pairs.

        ReleaseTexImageEXT
            1           CARD8           opcode (X assigned)
            1           16              GLX opcode (glXVendorPrivate)
            2           5               request length
            4           1331            vendor specific opcode
            4           CARD32          context tag
            4           GLX_DRAWABLE    drawable
            4           INT32           buffer
```

Errors

Additions to Chapter 2 of the OpenGL 1.2.1 Specification (OpenGL Operation)

None.

Additions to Chapter 3 of the OpenGL 1.2.1 Specification (Rasterization)

None.

Additions to Chapter 4 of the OpenGL 1.2.1 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 1.2.1 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 1.2.1 Specification (State and State Requests)

None.

Additions to the GLX Specification

Add to table 3.1, GLXFBConfig Attributes:

Attribute	Type	Notes
GLX_BIND_TO_TEXTURE_RGB_EXT	boolean	True if color buffers can be bound to RGB texture
GLX_BIND_TO_TEXTURE_RGBA_EXT	boolean	True if color buffers can be bound to RGBA texture
GLX_BIND_TO_MIPMAP_TEXTURE_EXT	boolean	True if color buffers can be bound to textures with multiple levels
GLX_BIND_TO_TEXTURE_TARGETS_EXT	bitmask	Bitmask of texture targets color buffers can be bound to
GLX_Y_INVERTED_EXT	boolean	True if the drawable's framebuffer is y-inverted. This can be used to determine if y-inverted texture coordinates need to be used when texturing from this drawable when it is bound to a texture target.

Additions to table 3.4, Default Match Criteria for GLXFBConfig attributes:

Attribute	Default	Selection Criteria	Priority
GLX_BIND_TO_TEXTURE_RGB_EXT	GLX_DONT_CARE	Exact	
GLX_BIND_TO_TEXTURE_RGBA_EXT	GLX_DONT_CARE	Exact	
GLX_BIND_TO_MIPMAP_TEXTURE_EXT	GLX_DONT_CARE	Exact	
GLX_BIND_TO_TEXTURE_TARGETS_EXT	-	Mask	
GLX_Y_INVERTED_EXT	GLX_DONT_CARE	Exact	

Modifications to 3.3.3, "Configuration Management"

Add after paragraph 17 in the description of FBConfigs:

GLX_Y_INVERTED_EXT is a boolean describing the memory layout used for drawables created with the GLXFBConfig. The attribute is True if the drawable's framebuffer will be y-inverted. This can be used to determine if y-inverted texture coordinates need to be used when texturing from this drawable when it is bound to a texture target.

Modifications to 3.3.5, "Offscreen Rendering"

Modify paragraph 3 of the description of glXCreatePixmap:

<attrib_list> specifies a list of attributes for the pixmap. The list has the same structure as described for glXChooseFBConfig. Currently the following attributes can be specified in attrib_list: GLX_TEXTURE_FORMAT_EXT, GLX_TEXTURE_TARGET_EXT, GLX_MIPMAP_TEXTURE_EXT, attrib_list may be NULL or empty (first attribute of None), in which case all attributes assume their default values as described below.

GLX_TEXTURE_FORMAT_EXT describes the texture format this pixmap can be bound to. Valid values are GLX_TEXTURE_FORMAT_RGB_EXT, GLX_TEXTURE_FORMAT_RGBA_EXT, and GLX_TEXTURE_FORMAT_NONE_EXT.

GLX_TEXTURE_TARGET_EXT can be set to GLX_TEXTURE_1D_EXT, GLX_TEXTURE_2D_EXT, or GLX_TEXTURE_RECTANGLE_EXT; it indicates the type of texture that will be created when GLX_TEXTURE_FORMAT_EXT is not GLX_TEXTURE_FORMAT_NONE_EXT. The default value of GLX_TEXTURE_TARGET_EXT depends on the capabilities in <config> and the dimensions of the pixmap.

```
    If <config> has GLX_TEXTURE_2D_BIT set and one or more of the following is
    true:

        * GLX_TEXTURE_RECTANGLE_BIT_EXT is not set in <config>

        * GL_ARB_texture_non_power_of_two is supported

        * the pixmap's width and height are powers of 2

    the default value for GLX_TEXTURE_TARGET_EXT is GLX_TEXTURE_2D_EXT.

    Otherwise, the first supported target is chosen in this order:
    GLX_TEXTURE_RECTANGLE_EXT, GLX_TEXTURE_1D_EXT.

    GLX_MIPMAP_TEXTURE_EXT indicates that storage for mipmaps should be
    allocated.  Space for mipmaps will be set aside if GLX_TEXTURE_FORMAT_EXT
    is not GLX_TEXTURE_FORMAT_NONE_EXT and GLX_MIPMAP_TEXTURE_EXT is TRUE.
    The default value is FALSE.

    Modify paragraph 5 of the description of glXCreatePixmap:

    ...If <pixmap> is not a valid Pixmap XID, then a BadPixmap error is
    generated.  A BadConfig error is generated if any of the following
    conditions are true:

        * GLX_TEXTURE_FORMAT_EXT is GLX_TEXTURE_FORMAT_RGB_EXT and
          <config> does not have GLX_BIND_TO_TEXTURE_RGB set to TRUE.

        * GLX_TEXTURE_FORMAT_EXT is GLX_TEXTURE_FORMAT_RGBA_EXT and
          <config> does not have GLX_BIND_TO_TEXTURE_RGBA set to TRUE.

        * GLX_MIPMAP_TEXTURE_EXT is set to TRUE and <config> does not
          have GLX_BIND_TO_MIPMAP_EXT set to TRUE.

        * GLX_TEXTURE_TARGET_EXT is set to GLX_TEXTURE_1D_EXT
          and <config> does not have GLX_TEXTURE_1D_BIT_EXT set.

        * GLX_TEXTURE_TARGET_EXT is set to GLX_TEXTURE_2D_EXT
          and <config> does not have GLX_TEXTURE_2D_BIT_EXT set.

        * GLX_TEXTURE_TARGET_EXT is set to GLX_TEXTURE_RECTANGLE_EXT
          and <config> does not have GLX_TEXTURE_RECTANGLE_BIT_EXT set.

    A BadValue error is generated if GLX_TEXTURE_FORMAT_EXT is not
    GLX_TEXTURE_FORMAT_NONE_EXT and the width or height of <pixmap> are
    incompatible with the specified value of GLX_TEXTURE_TARGET_EXT on this
    implementation. (e.g., the pixmap size is not a power of 2 and
    GL_ARB_texture_rectangle is not supported).

    Modify paragraph 1 of the description of glXDestroyPixmap:

    ...The storage for the GLX pixmap will be freed when it is not current
    to any client and all color buffers that are bound to a texture object
    have been released.
```

Modifications to seciton 3.3.6, "Querying Attributes"

Modify paragraph 1 of the description of glXQueryDrawable:

...<attribute> must be set to one of GLX_WIDTH, GLX_HEIGHT, GLX_PRESERVED_CONTENTS, GLX_LARGEST_PBUFFER, GLX_FBCONFIG_ID, GLX_TEXTURE_FORMAT_EXT, GLX_TEXTURE_TARGET_EXT or GLX_MIPMAP_TEXTURE_EXT or a BadValue error is generated.

Modify paragraph 3 of the description of glXQueryDrawable:

...If <draw> is a GLXWindow or GLXPixmap and <attribute> is set to GLX_PRESERVED_CONTENTS or GLX_LARGEST_PBUFFER, or if <draw> is a GLXWindow or GLXPbuffer and <attribute> is set to GLX_TEXTURE_FORMAT_EXT, GLX_TEXTURE_TARGET_EXT, or GLX_MIPMAP_TEXTURE_EXT, the contents of <value> are undefined.

Add a new section 3.3.6.1, "Texturing From Drawables"

The command

```
    void glXBindTexImageEXT (Display *dpy,
                             GLXDrawable draw,
                             int buffer,
                             int *attrib_list);
```

defines a one- or two-dimensional texture image. The texture image is taken from <buffer> and need not be copied. The texture target, the texture format, and the size of the texture components are derived from attributes of <draw>.

The drawable attribute GLX_TEXTURE_FORMAT_EXT determines the base internal format of the texture. The component sizes are also determined by drawable attributes as shown in table 3.4a.

Add new table 3.4a: Size of texture components:

Texture component	Size
R	GLX_RED_SIZE
G	GLX_GREEN_SIZE
B	GLX_BLUE_SIZE
A	GLX_ALPHA_SIZE

The texture target is derived from the GLX_TEXTURE_TARGET_EXT attribute of <draw>. If the texture target for the drawable is GLX_TEXTURE_2D_EXT or GLX_TEXTURE_RECTANGLE_EXT, then buffer defines a 2D texture for the current 2D or rectangle texture object respectively; if the texture target is GLX_TEXTURE_1D_EXT, then buffer defines a 1D texture for the current 1D texture object.

If <buffer> is not one of GLX_FRONT_LEFT_EXT, GLX_FRONT_RIGHT_EXT, GLX_BACK_LEFT_EXT, GLX_BACK_RIGHT_EXT, or GLX_AUX0_EXT through GLX_AUXn_EXT, where n is one less than the number of AUX buffers supported by the FBConfig used to create <draw>, or if the requested buffer is missing, a BadValue error is generated.

```
    <attrib_list> specifies a list of attributes for the texture.  The list has
    the same structure as described for glXChooseFBConfig.  If <attrib_list> is
    NULL or empty (first attribute of None), then all attributes assume their
    default values.  <attrib_list> must be NULL or empty.

    If <dpy> and <draw> are the display and drawable for the calling thread's
    current context, glXBindTexImageEXT performs an implicit glFlush.

    The contents of the texture after the drawable has been bound are defined
    as the result of all rendering that has completed before the call to
    glXBindTexImageEXT.  In other words, the results of any operation which
    has caused damage on the drawable prior to the glXBindTexImageEXT call
    will be represented in the texture.

    Rendering to the drawable while it is bound to a texture will leave the
    contents of the texture in an undefined state.  However, no
    synchronization between rendering and texturing is done by GLX.  It is
    the application's responsibility to implement any synchronization
    required.

    If a texture object is deleted before glXReleaseTexImageEXT is called,
    the color buffer is released.

    It is not an error to call TexImage2D, TexImage1D, CopyTexImage1D, or
    CopyTexImage2D to replace an image of a texture object that has a color
    buffer bound to it.  However, these calls will cause the color buffer to be
    released and new memory to be allocated for the texture.  Note that the
    color buffer is released even if the image that is being defined is a mipmap
    level that was not defined by the color buffer.  GenerateMipmapEXT is an
    exception.  GenerateMipmapEXT can be used to define mipmap levels for
    drawables that have been created with GLX_MIPMAP_TEXTURE_EXT set.  Calling
    GenerateMipmapEXT on a drawable that was created without
    GLX_MIPMAP_TEXTURE_EXT is undefined.

    The results of calling glXBindTexImageEXT when GENERATE_MIPMAP_SGIS is TRUE
    are undefined.

    If glXBindTexImageEXT is called and the drawable attribute
    GLX_TEXTURE_FORMAT_EXT is GLX_TEXTURE_FORMAT_NONE_EXT, then a BadMatch
    error is generated.

    Currently, only pixmaps can be bound to textures.  If <draw> is not a
    valid GLXPixmap, then a GLXBadPixmap error is generated.

    glXBindTexImageEXT is ignored if there is no current GLX rendering context.

    To release a color buffer that is being used as a texture, call

       void glXReleaseTexImageEXT (Dislpay *dpy, GLXDrawable draw, int buffer);

    <buffer> must be one of GLX_FRONT_LEFT_EXT, GLX_FRONT_RIGHT_EXT,
    GLX_BACK_LEFT_EXT, GLX_BACK_RIGHT_EXT, and GLX_AUX0_EXT through
    GLX_AUXn_EXT, where n is one less than the number of AUX buffers
    supported by the FBConfig used to create <draw> or a BadValue error
    is generated.

    The contents of the color buffer are unaffected by glXReleaseTexImageEXT.
    If the specified color buffer is no longer bound to a texture (e.g.,
    because the texture object was deleted), then glXReleaseTexImageEXT has no
    effect; no error is generated.

    When a color buffer is released (e.g., by calling glXReleaseTexImageEXT or
    implicitly by calling a routine such as TexImage2D), all textures that were
    defined by the color buffer become NULL.

    If glXReleaseTexImageEXT is called and the drawable attribute
    GLX_TEXTURE_FORMAT_EXT is GLX_TEXTURE_FORMAT_NONE_EXT, then a BadMatch
    error is generated.

    Currently, only pixmaps can be bound to textures.  If <draw> is not a
    valid GLXPixmap, then a GLXBadPixmap error is generated.
```

Usage Examples

```
    Example 1: Bind redirected window to texture:

    XGetWindowAttributes (display, window, &attrib);

    visualid = XVisualIDFromVisual (attrib.visual);

    fbconfigs = glXGetFBConfigs (display, screen, &nfbconfigs);
    for (i = 0; i < nfbconfigs; i++)
    {
        visinfo = glXGetVisualFromFBConfig (display, fbconfigs[i]);
        if (!visinfo || visinfo->visualid != visualid)
            continue;

        glXGetFBConfigAttrib (display, fbconfigs[i], GLX_DRAWABLE_TYPE, &value);
        if (!(value & GLX_PIXMAP_BIT))
            continue;

        glXGetFBConfigAttrib (display, fbconfigs[i],
                              GLX_BIND_TO_TEXTURE_TARGETS_EXT,
                              &value);
        if (!(value & GLX_TEXTURE_2D_BIT_EXT))
            continue;

        glXGetFBConfigAttrib (display, fbconfigs[i],
                              GLX_BIND_TO_TEXTURE_RGBA_EXT,
                              &value);
        if (value == FALSE)
        {
            glXGetFBConfigAttrib (display, fbconfigs[i],
                                  GLX_BIND_TO_TEXTURE_RGB_EXT,
                                  &value);
            if (value == FALSE)
                continue;
        }

        glXGetFBConfigAttrib (display, fbconfigs[i],
                              GLX_Y_INVERTED_EXT,
                              &value);
        if (value == TRUE)
        {
            top = 0.0f;
            bottom = 1.0f;
        }
        else
        {
            top = 1.0f;
            bottom = 0.0f;
        }

        break;
    }

    if (i == nfbconfigs)
        /* error 1 */
    pixmap = XCompositeNameWindowPixmap (display, window);
    pixmapAttribs = { GLX_TEXTURE_TARGET_EXT, GLX_TEXTURE_2D_EXT,
                      GLX_TEXTURE_FORMAT_EXT, GLX_TEXTURE_FORMAT_RGBA_EXT,
                      None };
    glxpixmap = glXCreatePixmap (display, fbconfigs[i], pixmap, pixmapAttribs);

    glGenTextures (1, &texture);
    glBindTexture (GL_TEXTURE_2D, texture);

    glXBindTexImageEXT (display, glxpixmap, GLX_FRONT_LEFT_EXT, NULL);

    glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

    /* draw using pixmap as texture */
    glBegin (GL_QUADS);

    glTexCoord2d (0.0f, bottom);
    glVertex2d (0.0f, 0.0f);

    glTexCoord2d (0.0f, top);
    glVertex2d (0.0f, 1.0f);

    glTexCoord2d (1.0f, top);
    glVertex2d (1.0f, 1.0f);

    glTexCoord2d (1.0f, bottom);
    glVertex2d (1.0f, 0.0f);

    glEnd ();

    glXReleaseTexImageEXT (display, glxpixmap, GLX_FRONT_LEFT_EXT);
```

Version History

```
    1. 26 Nov 2005 - DavidR
        Initial version
    2. 01 Dec 2005 - JamesJ
        -Adapted spec language from draft version of GLX_ARB_render_texture.
        -Added glXDrawableAttribute to set attributes.
        -Modified glXBindTexImageEXT to take an attrib_list parameter.
        -Added support for cubemap and 1D texture targets.
        -Added attribute to set the texture target when creating the
         drawable.
        -Updated the issues section.
        -Added mipmap support.  Support is not required.
        -Specified results of texturing from a drawable when it has been
         rendered to while bound to a texture as undefined until
         glXReleaseTexImageEXT has been called.  Allows implementations
         that need to perform an implicit copy after rendering occurs
         to be compatible with this specification.
    3. 04 Dec 2005 - DavidR
        -Changed name to GLX_EXT_texture_from_pixmap.
        -Changed spec regarding what happens when a pixmap that is bound
         to a texture is rendered to. Having textures be undefined once
         they are rendered to makes it useless for a compositing manager,
         which is a major use case for this extension.
        -Added support for not specifying texture target when creating a
         pixmap. Allows implementations to select whatever target it
         finds most suitable.
    4. 05 Dec 2005 - JamesJ
        -Changed the default value of GLX_TEXTURE_TARGET_EXT from
         GLX_NO_TEXTURE_EXT to something usable.  Eliminated
         GLX_NO_TEXTURE_EXT.
        -Eliminated GLX_TEXTURE_NONE_EXT.
        -Removed language referring to sharing of color buffers when
         pixmaps are bound to textures.
        -Updated issues.

    5. 13 Dec 2005 - JamesJ
        -Removed cube map support and rendering to multiple mipmap
         levels support.

    6. 20 Jan 2006 - JamesJ
        -Specified textures are y-inverted.

    7. 23 Jan 2006 - AaronP
        -Fix typos, make some things clearer.  Replace ocurrences of "released
         back to the drawable" with "released".

    8. 01 Feb 2006 - AndyR
        -Fix minor typos.

    9. 03 Feb 2006 - JamesJ
        -Added some new issues and their resolutions.
        -Finalized some issues that had been in discussion.
        -Made drawable y-inversion a queryable attribute of the drawable.
        -Moved detailed explanation of y-inverted addressing to the issues
         section
        -Updated example to demonstrate proper use of the y-inverted
         attribute.

    10. 06 Feb 2006 - DavidR
        -Made GLX_Y_INVERTED_EXT an FBConfig attribute instead of a drawable
         attribute.
        -Removed GLX_TEXTURE_CUBE_MAP_EXT.
        -Fix minor typo.

    11. 07 Feb 2006 - JamesJ
        -Added description of GLX_Y_INVERTED_EXT GLXFBConfig attribute, based
         on description of the drawable attribute of the same name from
         and earlier version of the specification.
        -Removed language requiring applications to re-bind a pixmap to a
         texture to gurantee contents of the texture are updated after a
         pixmap has been rendered to.
        -Added Aaron Plattner and Andy Ritger to contributors section.

    12. 14 Feb 2006 - JamesJ
        -Disallowed rendering to a drawable while it is bound as a texture
         and defined the exact contents of a texture after a drawable has
         been bound to it.
    13. 09 Mar 2006 - JamesJ
        -Add a context tag member to the vendor private requests.  This field
         is part of the vendor private header, and is needed to specify which
         context the BindTexImageEXT and ReleaseTexImageEXT requests correspond
         to.
        -Changed texture target bitfield values to not skip numbers removed in
         earlier updates.

    14. 13 Mar 2006 - JamesJ
        -Only require GLX_SGIX_fbconfig + GLX 1.2.
        -Clarify language regarding the result of rendering to drawables bound
         to textures.
        -Added GLX_FRONT_EXT and GLX_BACK_EXT tokens.

    15. 18 Apr 2006 - JamesJ
        -Allocated enum values and opcodes.
        -Require GLX 1.3.  GLX_SGIX_fbconfig doesn't allow creating pixmaps
         with attributes.
        -Added more arguments for not supporting rendering to multiple levels
         of a texture with this extension.
        -Fixed the inconsistencies in the return type of glXBindTexImageEXT
         and glXReleaseTexImageEXT.  It is now listed as void throughout.

    16. 12 Sep 2006 - JamesJ
        -Fix ordering of GLX protocol
```

Name

    NV_swap_group

Name Strings

    GLX_NV_swap_group

Status

    Shipping since 2003 on Quadro GPUs with framelock support

Version

    Date: 02/20/2008   Revision: 1.0

Number

    350

Dependencies

    Written based on the wording of the GLX_SGIX_swap_group and
    GLX_SGIX_swap_barrier specifications.

    SGIX_swap_control affects the definition of this extension

Overview

    This extension provides the capability to synchronize the buffer
    swaps of a group of OpenGL windows. A swap group is created, and
    windows are added as members to the swap group. Buffer swaps to
    members of the swap group will then take place concurrently.

    This extension also provides the capability to sychronize the buffer
    swaps of different swap groups, which may reside on distributed
    systems on a network. For this purpose swap groups can be bound to
    a swap barrier.

    This extension extends the set of conditions that must be met before
    a buffer swap can take place.

Issues

    An implementation can not guarantee that the initialization of the swap
    groups or barriers will succeed because the state of the window system may
    restrict the usage of these features. Once a swap group or barrier has
    been sucessfully initialized, the implementation can only guarantee to
    sustain swap group functionality as long as the state of the window system
    does not restrict this. An example for a state that does typically not
    restrict swap group usage is the use of one fullscreen sized window per
    desktop.

New Procedures and Functions

```
    Bool glXJoinSwapGroupNV(Display *dpy,
                            GLXDrawable drawable,
                            GLuint group);

    Bool glXBindSwapBarrierNV(Display *dpy,
                              GLuint group,
                              GLuint barrier);

    Bool glXQuerySwapGroupNV(Display *dpy,
                             GLXDrawable drawable,
                             GLuint *group,
                             GLuint *barrier);

    Bool glXQueryMaxSwapGroupsNV(Display *dpy,
                                 int screen,
                                 GLuint *maxGroups,
                                 GLuint *maxBarriers);

    Bool glXQueryFrameCountNV(Display *dpy,
                              int screen,
                              GLuint *count);

    Bool glXResetFrameCountNV(Display *dpy,
                              int screen);
```

New Tokens

none

Additions to the GLX Specification

Add to section 3.2.6, Double Buffering:

glXJoinSwapGroupNV adds <drawable> to the swap group specified by <group>. If <drawable> is already a member of a different group, it is implicitly removed from that group first. A swap group is specified as an integer value between 0 and the value returned in <maxGroups> by glXQueryMaxSwapGroupsNV. If <group> is zero, the drawable is unbound from its current group, if any. If <group> is larger than <maxGroups>, glXJoinSwapGroupNV fails.

glXJoinSwapGroupNV returns True if <drawable> has been successfully bound to <group> and False if it fails.

glXBindSwapBarrierNV binds the swap group specified by <group> to <barrier>. <barrier> is an integer value between 0 and the value returned in <maxBarriers> by glXQueryMaxSwapGroupsNV. If <barrier> is zero, the group is unbound from its current barrier, if any. If <barrier> is larger than <maxBarriers>, glXBindSwapBarrierNV fails. Subsequent buffer swaps for that group will be subject to this binding, until the group is unbound from <barrier>.

glXBindSwapBarrierNV returns True if <group> has been successfully bound to <barrier> and False if it fails.

glXQuerySwapGroupNV returns in <group> and <barrier> the group and barrier currently bound to <drawable,>, if any.

glXQuerySwapGroupNV returns True if <group> and <barrier> could be successfully queried for <drawable> and False if it fails. If it fails, the values of <group> and <barrier> are undefined.

glXQueryMaxSwapGroupsNV returns in <maxGroups> and <maxBarriers> the maximum number of swap groups and barriers supported by an implementation which drives <screen> and <dpy>.

glXQueryMaxSwapGroupsNV returns True if <maxGroups> and <maxBarriers> could be successfully queried for <screen> and <dpy>, and False if it fails. If it fails, the values of <maxGroups> and <maxBarriers> are undefined.

Before a buffer swap can take place, a set of conditions must be satisfied. The conditions are defined in terms of the notions of when a drawable is ready to swap and when a group is ready to swap.

GLX drawables except windows are always ready to swap.

When a window is unmapped, it is always ready.

A window is ready when all of the following are true:

1. A buffer swap command has been issued for it.

2. Its swap interval has elapsed.

A group is ready when the following is true:

1. All windows in the group are ready.

All of the following must be satisfied before a buffer swap for a window can take place:

1. The window is ready.

2. If the window belongs to a group, the group is ready.

3. If the window belongs to a group and that group is bound to a barrier, all groups using that barrier are ready.

Buffer swaps for all windows in a swap group will take place concurrently after the conditions are satisfied for every window in the group.

Buffer swaps for all groups using a barrier will take place concurrently after the conditions are satisfied for every window of every group using the barrier, if and only if the vertical retraces of the screens of all the groups are synchronized. If they are not synchronized, there is no guarantee of concurrency between groups.

An implementation may support a limited number of swap groups and barriers, and may have restrictions on where the users of a barrier can reside. For example, an implementation may allow the users to reside on different display devices or even hosts.

    An implementation may return zero for any of <maxGroups> and
    <maxBarriers> returned by glXQueryMaxSwapGroupsNV if swap groups or
    barriers are not available in that implementation or on that host.

    The implementation provides a universal counter, the so called
    frame counter, among all systems that are locked together by swap
    groups/barriers. It is based on the internal synchronization
    signal which triggers the buffer swap.

    glXQueryFrameCountNV returns in <count> the current frame counter
    for <swapGroup>.

    glXQueryFrameCountNV returns TRUE if the frame counter could be
    successfully retrieved. Otherwise it returns FALSE.

    glXResetFrameCountNV resets the frame counter of <swapGroup> to zero.

    glXResetFrameCountNV returns TRUE if the frame counter could be
    successfully reset, otherwise it returns FALSE. In a system that
    has an NVIDIA framelock add-on adapter installed and enabled,
    glXResetFrameCountNV will only succeed when the framelock is
    configured as a Master system.

    glXJoinSwapGroupNV, glXBindSwapBarrierNV, glXQuerySwapGroupNV,
    glXQueryMaxSwapGroupsNV, glXQueryFrameCountNV and
    glXResetFrameCountNV are part of the X stream.

Errors

    glXJoinSwapGroupNV, glXQuerySwapGroupNV and glXQueryMaxSwapGroupsNV
    generate GLXBadDrawable if <drawable> is an invalid GLX drawable.

New State

    None

New Implementation Dependent State

    None

Name

    NV_video_output

Name Strings

    GLX_NV_video_output

Status

    Shipping since 2004 for NVIDIA Quadro SDI (Serial Digital Interface)

Version

```
    Last Modified:      $Date: 2008/02/20 $
    NVIDIA Revision:    $Revision: #5 $
```

Number

    348

Dependencies

```
    OpenGL 1.1 is required.
    GLX 1.3 is required.
```

Overview

    This extension permits a color and or depth buffer of a pbuffer to
    be used for rendering and subsequent video output. After a pbuffer
    has been bound to a video device, subsequent color and or depth
    rendering into that buffer may be displayed on the video output.

    This is intended for use with NVIDIA products such as the Quadro FX
    4000 SDI.

Issues

```
 1. Should the new pbuffer attributes be available through GL queries?

    No, like other pbuffer attributes you need to query them through the
    window system extension. This extension does not make any changes to
    OpenGL.

 2. Should glXSendPbufferToVideoNV require that the pbuffer be current?
```

Implementation Notes

```
 1. Any created pbuffers must be the same resolution as that specified
    by the state of the video output device.  The current state of the
    video output device can be queried via the NV-CONTROL X extension.

 2. Applications may use a single pbuffer or a collection of pbuffers
    to send frames/fields to a video device.  In the first case, an
    application should block on the call to glXSendPbufferToVideoNV() to
    ensure synchronization.  In the second case, an application should
```

    utilize glXGetVideoInfoNV() in order to query vblank and
    buffer counters for synchronization.

Intended Usage

    1) Configure the video output device via the NV-CONTROL X extension.

    2) Use glXGetFBConfigs or glXChooseFBConfig to find a suitable
       FBConfig for rendering images. GLX_DRAWABLE_TYPE must have
       GLX_PBUFFER_BIT set. The per-component pixel depth of the pbuffer
       must be equal to or greater than the per-component depth of the
       video output.

    3) Create a GLXPbuffer for each stream of video by calling
       glXCreatePbuffer. Set the width and height for each GLXPbuffer
       to match that of the intended video output device.

    4) Call glXGetVideoDeviceNV to retrieve the handles for all
       video devices available. A video device handle is required
       for each video stream. glXGetVideoDeviceNV will lock the
       video device for exclusive use by this GLX client. The NV-CONTROL
       X extension will not be able to update video out attributes until
       the video device is released with glXReleaseVideoDeviceNV.

    5) Call glXBindVideoImageNV to bind each GLXPbuffer to a
       corresponding video device handle. Multiple pbuffers can
       be bound, at the same time, to the same video device.

    6) Render the current frame/field for each stream to one of the bound
       GLXPbuffers. Once rendering is complete, call
       glXSendPbufferToVideoNV to send each frame/field to the video
       device.

    7) Render subsequent video frames or fields calling
       glXSendPbufferToVideoNV() at the completion of rendering for
       each frame/field.

    8) Call glXReleaseVideoImageNV to unbind each GLXPbuffer
       from its associated video device.

    9) Call glXReleaseVideoDeviceNV to release the video device.

New Types

```
    /*
     * GLXVideoDeviceNV is an opaque handle to a video device.
     */
    typedef struct unsigned int GLXVideoDeviceNV;
```

New Procedures and Functions

```
    int glXGetVideoDeviceNV(Display *dpy, int screen, int numVideoDevices,
                            GLXVideoDeviceNV *pVideoDevice);

    int glXReleaseVideoDeviceNV(Display *dpy, int screen,
                                GLXVideoDeviceNV VideoDevice);
    int glXBindVideoImageNV(Display *dpy, GLXVideoDeviceNV VideoDevice,
                            GLXPbuffer pbuf, int iVideoBuffer);

    int glXReleaseVideoImageNV(Display *dpy, GLXPbuffer pbuf);

    int glXSendPbufferToVideoNV(Display *dpy, GLXPbuffer pbuf,
                                int iBufferType,
                                unsigned long *pulCounterPbuffer,
                                GLboolean bBlock);

    int glXGetVideoInfoNV(Display *dpy, int screen,
                          GLXVideoDeviceNV VideoDevice,
                          unsigned long *pulCounterOutputPbuffer,
                          unsigned long *pulCounterOutputVideo);
```

New Tokens

```
    Accepted by the <iVideoBuffer> parameter of glXBindVideoImageNV:

        GLX_VIDEO_OUT_COLOR_NV                          0x20C3
        GLX_VIDEO_OUT_ALPHA_NV                          0x20C4
        GLX_VIDEO_OUT_DEPTH_NV                          0x20C5
        GLX_VIDEO_OUT_COLOR_AND_ALPHA_NV                0x20C6
        GLX_VIDEO_OUT_COLOR_AND_DEPTH_NV                0x20C7

    Accepted by the <iBufferType> parameter of glXSendPbufferToVideoNV:

        GLX_VIDEO_OUT_FRAME_NV                          0x20C8
        GLX_VIDEO_OUT_FIELD_1_NV                        0x20C9
        GLX_VIDEO_OUT_FIELD_2_NV                        0x20CA
        GLX_VIDEO_OUT_STACKED_FIELDS_1_2_NV             0x20CB
        GLX_VIDEO_OUT_STACKED_FIELDS_2_1_NV             0x20CC
```

Additions to Chapter 2 of the OpenGL 1.2.1 Specification (OpenGL Operation)

None.

Additions to Chapter 3 of the OpenGL 1.2.1 Specification (Rasterization)

None.

Additions to Chapter 4 of the OpenGL 1.2.1 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 1.2.1 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 1.2.1 Specification (State and State Requests)

None.

Additions to the GLX 1.3 Specification

[Add new section, Video Out]

Video out functions permit color and depth buffers from a pbuffer to be sent to a video output device.

The command

```
    int glXGetVideoDeviceNV(Display *dpy, int screen, int numVideoDevices,
                            GLXVideoDeviceNV *pVideoDevice);
```

fills in the array <pVideoDevice> with up to <numVideoDevices> handles to the available video devices. <numVideoDevices> must be non-negative, and <pVideoDevice> must not be NULL.

It is not an error if the number of available video devices is larger that <numVideoDevices>; in that case the first <numVideoDevices> device handles are returned. It is an error if <numVideoDevices> is larger than the number of available video devices. The order of devices returned in <pVideoDevice> is implementation dependent.

If glXGetVideoDeviceNV succeeds, 0 is returned. Otherwise, a non-zero error code is returned.

The command

```
    int glXReleaseVideoDeviceNV(Display *dpy, int screen,
                                GLXVideoDeviceNV VideoDevice);
```

releases all resources associated with <VideoDevice>.

If glXReleaseVideoDeviceNV succeeds, 0 is returned. Otherwise, a non-zero error code is returned.

The command

```
    int glXBindVideoImageNV(Display *dpy, GLXVideoDeviceNV VideoDevice,
                            GLXPbuffer pbuf, int iVideoBuffer);
```

binds <pbuf> to <VideoDevice> for subsequent scanout where <iVideoBuffer> specifies that <pbuf> contains color, alpha and/or depth data. Valid values for <iVideoBuffer> are:

GLX_VIDEO_OUT_COLOR_NV	0x20C3
GLX_VIDEO_OUT_ALPHA_NV	0x20C4
GLX_VIDEO_OUT_DEPTH_NV	0x20C5
GLX_VIDEO_OUT_COLOR_AND_ALPHA_NV	0x20C6
GLX_VIDEO_OUT_COLOR_AND_DEPTH_NV	0x20C7

<pbuf> cannot be None, and <VideoDevice> must be a VideoDevice returned by glXGetVideoDeviceNV().

```
    A pbuffer can only be bound to one GLXVideoDeviceNV at a time.
    If <pbuf> is already bound to a different GLXVideoDeviceNV, then
    glXBindVideoImageNV will fail.


    If glXBindVideoImageNV succeeds, 0 is returned.  Otherwise,
    a non-zero error code is returned.


    The command

        int glXReleaseVideoImageNV(Display *dpy, GLXPbuffer pbuf);

    releases <pbuf> from a previously bound video device.  <pbuf> may
    not be None.

    If glXReleaseVideoImageNV succeeds, 0 is returned.  Otherwise,
    a non-zero error code is returned.


    The command

        int glXSendPbufferToVideoNV(Display *dpy, GLXPbuffer pbuf,
                                    int iBufferType,
                                    unsigned long *pulCounterPbuffer,
                                    Bool bBlock);

    indicates that rendering to the <pbuf> is complete and that the
    completed frame/field contained with <pbuf> is ready for scan out by
    the video device where <iBufferType> specifies that <pbuf> contains
    the first field, second field or a complete frame.  Valid values
    for <iBufferType> are:

        GLX_VIDEO_OUT_FRAME_NV                          0x20C8
        GLX_VIDEO_OUT_FIELD_1_NV                        0x20C9
        GLX_VIDEO_OUT_FIELD_2_NV                        0x20CA

    The color buffer controlled by glReadBuffer is used as the color
    buffer input to glXSendPbufferToVideoNV().  <pbuf> cannot be None.
    The <bBlock> argument specifies whether or not the call should
    block until scan out of the specified frame/field is complete.
    <pulCounterPbuffer> returns the total number of frames/fields sent
    to the video device.

    If glXSendPbufferToVideoNV succeeds, 0 is returned.  Otherwise,
    a non-zero error code is returned.


    The command

        int glXGetVideoInfoNV(GLXVideoDeviceNV VideoDevice,
                              unsigned long *pulCounterOutputPbuffer,
                              unsigned long *pulCounterOutputVideo);

    returns in <pulCounterOutputVideo> the absolute count of vertical
    blanks on <VideoDevice> since transfers were started while
```

    <pulCounterOutputPbuffer> returns the count of the current pbuffer
    being scanned out by <VideoDevice>.

    If glXGetVideoInfoNV succeeds, 0 is returned. Otherwise, a non-zero
    error code is returned.

GLX Protocol

    Six new GLX protocol commands are added.

GetVideoDeviceNV

```
        1       CARD8       opcode (X assigned)
        1       17          GLX opcode (glXVendorPrivateWithReply)
        2       5           request length
        4       1313        vendor specific opcode
        4                   unused
        4       CARD32      num_devices
        4       CARD32      screen
      =>
        1       CARD8       reply
        1                   unused
        2       CARD16      sequence number
        4       n           reply length
        4       CARD32      status
        4       CARD32      num_devices
        16                  unused
        4 * n   CARD32      video_device handles
```

    Where n is the number of device handles returned.

ReleaseVideoDeviceNV

```
        1       CARD8       opcode (X assigned)
        1       17          GLX opcode (glXVendorPrivateWithReply)
        2       5           request length
        4       1314        vendor specific opcode
        4                   unused
        4       CARD32      video_device
        4       CARD32      screen
      =>
        1       CARD8       reply
        1                   unused
        2       CARD16      sequence number
        4       0           reply length
        4       CARD32      status
        20                  unused
```

```
    BindVideoImageNV
        1       CARD8       opcode (X assigned)
        1       17          GLX opcode (glXVendorPrivateWithReply)
        2       6           request length
        4       1314        vendor specific opcode
        4                   unused
        4       GLX_PBUFFER pbuffer
        4       CARD32      video_device
        4       CARD32      video_buffer
      =>
        1       CARD8       reply
        1                   unused
        2       CARD16      sequence number
        4       0           reply length
        4       CARD32      status
        20                  unused

    ReleaseVideoImageNV
        1       CARD8       opcode (X assigned)
        1       17          GLX opcode (glXVendorPrivateWithReply)
        2       4           request length
        4       1315        vendor specific opcode
        4       GLX_PBUFFER pbuffer
      =>
        1       CARD8       reply
        1                   unused
        2       CARD16      sequence number
        4       0           reply length
        4       CARD32      status
        20                  unused

    SendPbufferToVideoNV
        1       CARD8       opcode (X assigned)
        1       17          GLX opcode (glXVendorPrivateWithReply)
        2       6           request length
        4       1316        vendor specific opcode
        4                   unused
        4       GLX_PBUFFER pbuffer
        4       CARD32      buffer_type
        1       BOOL        block
        1                   unused
        2                   unused
      =>
        1       CARD8       reply
        1                   unused
        2       CARD16      sequence number
        4       0           reply length
        4       CARD32      status
        4       CARD32      counter_pbuffer
        16                  unused
    GetVideoInfoNV
        1       CARD8       opcode (X assigned)
        1       17          GLX opcode (glXVendorPrivateWithReply)
        2       5           request length
        4       1317        vendor specific opcode
        4       CARD32      screen
        4       CARD32      video_device
      =>
        1       CARD8       reply
        1                   unused
        2       CARD16      sequence number
        4       0           reply length
        4       CARD32      status
        4       CARD32      counter_video
        4       CARD32      counter_pbuffer
        12                  unused
```

New State

    None

New Implementation Dependent State

    None

Name

    WGL_ARB_buffer_region

Name Strings

    WGL_ARB_buffer_region

Status

    Complete. Approved by ARB on 12/8/1999

Version

    Last Modified Date: December 10, 2000
    Intergraph Revision 1.0

Number

    ARB Extension #4

Dependencies

    The extension is written against the OpenGL 1.2.1 Specification
    although it should work on any previous OpenGL specification.

    The WGL_EXT_extensions_string extension is required.

Overview

    The buffer region extension is a mechanism that allows an area of
    an OpenGL window to be saved in off-screen memory for quick
    restores. The off-screen memory can either be frame buffer memory
    or system memory, although frame buffer memory might offer optimal
    performance.

    A buffer region can be created for the front color, back color,
    depth, and/or stencil buffer. Multiple buffer regions for the same
    buffer type can exist.

IP Status

    None

Issues

    1. Do we need the glBufferRegionEnabled call that is in the
       Kinetix extensions?

       The reason behind this function was so that a single driver
       could be used on adapters with various amounts of memory -- the
       extension would always be present but its use would depend on a
       separate call. The same functionality could be achieved by not
       advertising this extension or always returning a value of NULL
       from wglCreateBufferRegionARB.

```
    2. Should the width/height be specified on the create.

       Because applications create regions that are not used, it would
       be better to leave the size as a parameter on the save.

    3. Should information be added to the create to allow for layer
       support?

       Layer support has been added.

    4. Which DC gets used for buffer region operations?

       The DC that was allocated on the CreateBufferRegionARB call is
       saved and used for subsequent save and restore operations.  It
       must remain valid during the life of the buffer region.  This is
       analogous to the RC method for handling the DC.

    5. Does the driver do a flush before the save and restore?

       In keeping with the same paradigm as SwapBuffers, a flush will
       be made by the driver for the RC bound to the calling thread
       before the save and restore operations.

    6. Which coordinate system is used?

       The KTX_buffer_region and WIN_swap_hint extensions specify the
       (x,y) origin as the lower left corner of the rectangle.  This
       extension adopts the same philosophy.
```

New Procedures and Functions

```
    HANDLE wglCreateBufferRegionARB(HDC hDC,
                                     int iLayerPlane,
                                     UINT uType)

    VOID wglDeleteBufferRegionARB(HANDLE hRegion)

    BOOL wglSaveBufferRegionARB(HANDLE hRegion,
                                int x,
                                int y,
                                int width,
                                int height)

    BOOL wglRestoreBufferRegionARB(HANDLE hRegion,
                                   int x,
                                   int y,
                                   int width,
                                   int height,
                                   int xSrc,
                                   int ySrc)
```

New Tokens

    Accepted by the <uType> parameter of wglCreateBufferRegionARB is the
    bitwise OR of any of the following values:

```
      WGL_FRONT_COLOR_BUFFER_BIT_ARB          0x00000001
      WGL_BACK_COLOR_BUFFER_BIT_ARB           0x00000002
      WGL_DEPTH_BUFFER_BIT_ARB                0x00000004
      WGL_STENCIL_BUFFER_BIT_ARB              0x00000008
```

Additions to Chapter 2 of the OpenGL 1.2.1 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the OpenGL 1.2.1 Specification (Rasterization)

None

Additions to Chapter 4 of the OpenGL 1.2.1 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the OpenGL 1.2.1 Specification (Special Functions)

None

Additions to Chapter 6 of the OpenGL 1.2.1 Specification (State and State Requests)

None

Additions to Appendix A of the OpenGL 1.2.1 Specification (Invariance)

None

Additions to the GLX Specification

None

GLX Protocol

None

Additions to the WGL Specification

A buffer region can be created with wglCreateBufferRegionARB which returns a handle associated with the buffer region.

```
      HANDLE wglCreateBufferRegionARB(HDC hDC,
                                      INT iLayerPlane,
                                      UINT uType)
```

<hDC> specifies a device context for the device on which the buffer region is created. <iLayerPlane> specifies the layer. Positive values identify overlay planes, negative values identify underlay planes. A value of 0 identifies the main plane.

```
    <uType> is a bitwise OR of any of the following values indicating
    which buffers can be saved or restored.  Multiple bits can be set
    and may result in better performance if multiple buffers are saved
    or restored.

        WGL_FRONT_COLOR_BUFFER_BIT_ARB
        WGL_BACK_COLOR_BUFFER_BIT_ARB
        WGL_DEPTH_BUFFER_BIT_ARB
        WGL_STENCIL_BUFFER_BIT_ARB

    For stereo windows, WGL_FRONT_COLOR_BUFFER_BIT_ARB implies both the
    left and right front buffers, and WGL_BACK_COLOR_BUFFER_BIT_ARB
    implies both the left and right back buffers.

    When wglCreateBufferRegionARB fails to create a buffer region, a
    value of NULL is returned.  To get extended error information, call
    GetLastError.

    Image, depth, and stencil data can be saved into the buffer region
    by calling wglSaveBufferRegionARB.

      BOOL wglSaveBufferRegionARB(HANDLE hRegion,
                                  int x,
                                  int y,
                                  int width,
                                  int height)

    <hRegion> is a handle to a buffer region previously created with
    wglCreateBufferRegionARB.  The DC specified when the region was
    created is used as the device context specifying the window.

    <x> and <y> specify the window position for the source rectangle.
    <width> and <height> specify the width and height of the source
    rectangle.  Data outside the window for the specified rectangle is
    undefined.  The OpenGL coordinate system is used for specifying the
    rectangle (<x> and <y> specify the lower-left corner of the
    rectangle).

    If an RC is current to the calling thread, a flush will occur
    before the save operation.

    The saved buffer region area can be freed by calling
    wglSaveBufferRegionARB with <width> or <height> set to a value
    of 0.

    If the call to wglSaveBufferRegionARB is successful, a value of
    TRUE is returned.  Otherwise, a value of FALSE is returned.  To
    get extended error information, call GetLastError.
```

    A previously saved region can be restored (multiple times) with
    the wglRestoreBufferRegionARB function.

```
      BOOL wglRestoreBufferRegionARB(HANDLE hRegion,
                                     int x,
                                     int y,
                                     int width,
                                     int height,
                                     int xSrc,
                                     int ySrc)
```

    <hRegion> is a handle to a buffer region previously created with
    wglCreateBufferRegionARB. The DC specified when the region was
    created is used as the device context specifying the window.

    <x> and <y> specify the window position for the destination
    rectangle. <width> and <height> specify the width and height of
    the destination rectangle. The OpenGL coordinate system is used
    for specifying the rectangle (<x> and <y> specify the lower-left
    corner of the rectangle).

    <xSrc> and <ySrc> specify the position in the buffer region for
    the source of the data. Any portion of the rectangle outside of
    the saved region is ignored.

    If an RC is current to the calling thread, a flush will occur
    before the restore operation.

    If the call to wglRestoreBufferRegionARB is successful, a value of
    TRUE is returned. Otherwise, a value of FALSE is returned. To
    get extended error information, call GetLastError.

    The buffer region can be deleted with wglDeleteBufferRegionARB.

```
      VOID wglDeleteBufferRegionARB(HANDLE hRegion)
```

    <hRegion> is a handle to a buffer region previously created with
    wglCreateBufferRegionARB. Any saved data associated with <hRegion>
    is discarded. The DC used to create the region must still be valid
    for the delete to work.

Dependencies on WGL_EXT_extensions_string

    Because there is no way to extend wgl, these calls are defined in
    the ICD and can be called by obtaining the address with
    wglGetProcAddress. Because this extension is a WGL extension, it
    is not included in the GL_EXTENSIONS string. Its existence can be
    determined with the WGL_EXT_extensions_string extension.

Errors

    ERROR_NO_SYSTEM_RESOURCES is generated if memory cannot be
    allocated for storing the saved data.

    ERROR_INVALID_HANDLE is generated if <hRegion> is not a valid
    handle that was previously returned by wglCreateBufferRegionARB.

ERROR_INVALID_DATA is generated if <uType> is zero or includes an undefined bit.

ERROR_INVALID_DATA is generated if <width> or <height> is negative.

New State

None

New Implementation Dependent State

None

Conformance Test

1. Clear the window to blue.
2. Save an area of the window using wglSaveBufferRegionARB.
3. Clear the window to red.
4. Restore the area of the window using wglRestoreBufferRegionARB.
5. Verify that the area was restored.
6. Repeat for the depth buffer.
7. Repeat for the stencil buffer.
8. Repeat for image and depth buffer.

Revision History

```
12/10/99  1.0  ARB extension - based on the wgl_buffer_region
               extension.
```

Name

    WGL_ARB_extensions_string

Name Strings

    WGL_ARB_extensions_string

Status

    Complete. Approved by ARB on March 15, 2000

Version

    Last Modified Date: March 22, 2000
    Author Revision: 1.0

Number

    ARB Extension #8

Dependencies

    None

Overview

    This extension provides a way for applications to determine which
    WGL extensions are supported by a device. This is the foundation
    upon which other WGL extensions are built.

IP Status

    No issues.

Issues

1. Note that extensions that were previously advertised via glGetString (e.g., the swap interval extension) should continue to be advertised there so existing applications don't break. They should also be advertised via wglGetExtensionsStringARB so new applications can make one call to find out which WGL extensions are supported.

2. Should this function take an hdc? It seems like a good idea. At some point MS may want to incorporate this into OpenGL32. If they do this and and they want to support more than one ICD, then an HDC would be needed.

New Procedures and Functions

```
    const char *wglGetExtensionsStringARB(HDC hdc);
```

New Tokens

    None

Additions to Chapter 2 of the 1.2 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.2 Specification (Rasterization)

None

Additions to Chapter 4 of the 1.2 Specification (Per-Fragment Operations and the Frame buffer)

None

Additions to Chapter 5 of the 1.2 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

None

Additions to the WGL Specification

Advertising WGL Extensions

Applications should call wglGetProcAddress to see whether or not wglGetExtensionsStringARB is supported. If it is supported then it can be used to determine which WGL extensions are supported by the device.

```
const char *wglGetExtensionsStringARB(HDC hdc);

<hdc>   device context to query extensions for
```

If the function succeeds, it returns a list of supported extensions to WGL. Although the contents of the string is implementation specific, the string will be NULL terminated and will contain a space-separated list of extension names. (The extension names themselves do not contain spaces.) If there are no extensions then the empty string is returned.

If <hdc> does not indicate a valid device context then the function fails and the error ERROR_DC_NOT_FOUND is generated. If the function fails, the return value is NULL. To get extended error information, call GetLastError.

New State

None

New Implementation Dependent State

None

Revision History

```
    Changes from EXT_extension_string:

    Added hdc parameter to facilitate moving this function into OPENGL32
    Added WGL to name to avoid name collisions with GL and GLX
```

Name

    WGL_ARB_make_current_read

Name Strings

    WGL_ARB_make_current_read

Status

    Complete. Approved by ARB on March 15, 2000.

Version

```
    Last Modified Date: 03/22/2000
    Author Revision: 1.0

    Based on:  WGL_EXT_pbuffer specification
               Date: 3/1/1999   Version: 1.5
```

Number

    ARB Extension #10

Dependencies

    WGL_ARB_extensions_string is required.

Overview

```
    The association of a separate "read" and "draw" DC with the current
    context allows for preprocessing of image data in an "off screen"
    DC which is then read into a visible DC for final display.
```

New Procedures and Functions

```
    BOOL wglMakeContextCurrentARB(HDC hDrawDC,
                                  HDC hReadDC,
                                  HGLRC hglrc);

    HDC wglGetCurrentReadDCARB(VOID);
```

New Tokens

```
    New errors returned by GetLastError:

      ERROR_INVALID_PIXEL_TYPE_ARB              0x2043
      ERROR_INCOMPATIBLE_DEVICE_CONTEXTS_ARB    0x2054
```

Additions to Chapter 2 of the 1.2 GL Specification (OpenGL Operation)

    None

Additions to Chapter 3 of the 1.2 GL Specification (Rasterization)

    None

Additions to Chapter 4 of the 1.2 Specification (Per-Fragment Operations and the Frame buffer)

None

Additions to Chapter 5 of the 1.2 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

None

Additions to the GLX Specification

This specification is written for WGL.

GLX Protocol

This specification is written for WGL.

Additions to the WGL specification

The function wglMakeContextCurrentARB associates the context <hglrc> with the device <hDrawDC> for draws and the device <hReadDC> for reads. All subsequent OpenGL calls made by the calling thread are drawn on the device identified by <hDrawDC> and read on the device identified by <hReadDC>.

The <hDrawDC> and <hReadDC> parameters must refer to drawing surfaces supported by OpenGL. These parameters need not be the same <hdc> that was passed to wglCreateContext when <hglrc> was created. <hDrawDC> must have the same pixel format and be created on the same physical device as the <hdc> that was passed into wglCreateContext. <hReadDC> must be created on the same device as the <hdc> that was passed to wglCreateContext and it must support the same pixel type as the pixel format of the <hdc> that was passed to wglCreateContext.

If wglMakeContextCurrentARB is used to associate a different device for reads than for draws, the "read" device will be used for the following OpenGL operations:

1. Any pixel data that are sourced based on the value of READ_BUFFER. Note, that accumulation operations use the value of READ_BUFFER, but are not allowed when a different device context is used for reads. In this case, the accumulation operation will generate INVALID_OPERATION.

2. Any depth values that are retrieved by ReadPixels, CopyPixels, or any OpenGL extension that sources depth images from the frame buffer in the manner of ReadPixels and CopyPixels.

3. Any stencil values that are retrieved by ReadPixels, CopyPixels, or any OpenGL extension that sources stencil images from the framebuffer in the manner of ReadPixels and CopyPixels.

```
    These frame buffer values are taken from the surface associated with
    the device context specified by <hReadDC>.

    No error will be generated if the value of READ_BUFFER at the time
    the wglMakeContextCurrentARB call is made does not correspond to a
    valid color buffer in <hReadDC>.  Also, no error due to READ_BUFFER
    mismatch will be generated by subsequent calls to any of the
    operations enumerated above, but the pixels values used will be
    undefined until READ_BUFFER is set to a color buffer that is valid
    in the <hReadDC>.  Operations that query the value of READ_BUFFER
    (i.e., Get, PushAttrib) use the value set last in the context,
    independent of whether it is a valid buffer in <hReadDC>.

    Error conditions set by ReadBuffer and by the operations enumerated
    above are with respect to color and ancillary buffers available in
    <hReadDC> (i.e., ReadBuffer(BACK_BUFFER) will generate an error
    when <hReadDC> is single buffered, and so will an operation that
    tries to source stencil images when <hReadDC> does not have a
    stencil buffer).  When the read buffer is set implicitly via
    PopAttrib to a state not supported by the pixel format, an error
    may be generated.

    If wglMakeContextCurrentARB succeeds, the return value is TRUE.
    If the function fails, the return value is FALSE.  To get extended
    error information, call GetLastError.  Possible errors are as follows:

      ERROR_INVALID_PIXEL_FORMAT     The pixel format associated with
                                     <hDrawDC> does not match the pixel
                                     format associated with the render
                                     context.

      ERROR_INVALID_PIXEL_TYPE_ARB   The pixel type for <hReadDC> is
                                     different than the pixel type
                                     associated with the <hdc> that was
                                     passed to wglCreateContext.

      ERROR_INCOMPATIBLE_DEVICE_CONTEXTS_ARB
                                     The device contexts specified by
                                     <hReadDC> and <hDrawDC> are not
                                     compatible.  This can occur if the
                                     device contexts are managed by
                                     different drivers or possibly on
                                     different graphics adapters.

      ERROR_DC_NOT_FOUND             <hReadDC> or <hDrawDC> is not a valid
                                     device context.

      ERROR_NO_SYSTEM_RESOURCES      The device contexts specified by
                                     <hReadDC> and <hDrawDC> cannot exist
                                     in the framebuffer simultaneously.

    wglGetCurrentReadDC returns a handle to the "read" device context that
    is associated with the current OpenGL rendering context of the calling
    thread.  If the calling thread does not have a current context, the
    return value is NULL.
```

Dependencies on WGL_ARB_extensions_string

    Because there is no way to extend wgl, these calls are defined in
    the ICD and can be called by obtaining the address with
    wglGetProcAddress. Because this extension is a WGL extension, it
    is not included in the GL_EXTENSIONS string. Its existence can be
    determined with the WGL_ARB_extensions_string extension.

New State

    None

New Implementation Dependent State

   None

Conformance Testing

```
    1. Create two non-overlapping windows (windows 1 and 2).
    2. Create three contexts (context A, B, and C).
    3. Set context A to draw to window 1 and read from window 1.
    4. Set context B to draw to window 2 and read from window 1.
    5. Set context C to draw to window 2 and read from window 2.
    6. For a conformance test (TBD),
       a. Draw using context A.
       b. Blit from window to window using context B.
       c. Test conformance using context C.
    7. If pixel buffers are supported, repeat using a pixel buffer.
```

Revision History

```
    12/16/1999  0.1
        - First ARB draft based on the EXT specification.

    03/15/2000  0.2
        - Removed the changes to Chapter 4.
        - Added a discussion that accumulation operations may
          generate INVALID_OPERATION.
        - PopAttrib may (not will) generate an error.
        - Added an error if the read and draw DCs are not managed
          on the same driver.

    03/22/2000  1.0
        - Changed rendering context to device context.
        - Added the new error conditions values.
        - Approved by ARB: 10-0-0.
```

Name

    WGL_ARB_pbuffer

Name Strings

    WGL_ARB_pbuffer

Status

    Complete. Approved by ARB on March 15, 2000

Version

```
    Last Modified Date: 03/22/2000
    Author Revision: 1.0

    Based on:  WGL_EXT_pbuffer specification
               Date: 4/21/1999   Version 1.8
```

Number

    ARB Extension #11

Dependencies

```
    WGL_ARB_extensions_string is required.
    WGL_ARB_pixel_format is required.
    WGL_ARB_make_current_read affects the definition of this extension.
```

Overview

This extension defines pixel buffers (pbuffer for short). Pbuffers are additional non-visible rendering buffers for an OpenGL renderer. Pbuffers are equivalent to a window that has the same pixel format descriptor with the following exceptions:

1. There is no rendering to a pbuffer by GDI.

2. The pixel format descriptors used for a pbuffer can only be those that are supported by the ICD. Generic formats are not valid.

3. The allocation of a pbuffer can fail if there are insufficient resources (i.e., all the pbuffer memory has been allocated).

4. The pixel buffer might be lost if a display mode change occurs. A query is provided that can be called after a display mode change to determine the state of the pixel buffer.

The intent of the pbuffer semantics is to enable implementations to allocate pbuffers in non-visible frame buffer memory. These pbuffers are intended to be "static" resources in that a program will typically allocate them only once rather than as a part of its rendering loop. (Pbuffers should be deallocated when the program is no longer using them -- for example, if the program is iconified.)

The frame buffer resources that are associated with a pbuffer are also static and are deallocated when the pbuffer is destroyed or possibly when a display mode change occurs.

IP Status

TBD

Issues

1. Should the OPTIMUM width and heights and PBUFFER_LARGEST_ARB be taken out of the spec since they may be misleading or hard for some implementations to support?

   PBUFFER_LARGEST_ARB has been left in the extension. It was originally requested by an application. The OPTIMUM queries have been removed to match the GLX pixel buffer specification.

New Procedures and Functions

```
    DECLARE_HANDLE(HPBUFFERARB);

    HPBUFFERARB wglCreatePbufferARB(HDC hDC,
                                    int iPixelFormat,
                                    int iWidth,
                                    int iHeight,
                                    const int *piAttribList);

    HDC wglGetPbufferDCARB(HPBUFFERARB hPbuffer);

    int wglReleasePbufferDCARB(HPBUFFERARB hPbuffer,
                               HDC hDC);

    BOOL wglDestroyPbufferARB(HPBUFFERARB hPbuffer);

    BOOL wglQueryPbufferARB(HPBUFFERARB hPbuffer,
                            int iAttribute,
                            int *piValue);
```

New Tokens

Accepted by the <attribute> parameter of wglChoosePixelFormatEXT:

```
      WGL_DRAW_TO_PBUFFER_ARB              0x202D
```

Accepted by the <attribute> parameter of wglGetPixelFormatAttribivEXT, and wglGetPixelFormatAttribfvEXT:

```
      WGL_DRAW_TO_PBUFFER_ARB              0x202D
      WGL_MAX_PBUFFER_PIXELS_ARB           0x202E
      WGL_MAX_PBUFFER_WIDTH_ARB            0x202F
      WGL_MAX_PBUFFER_HEIGHT_ARB           0x2030
```

```
    Accepted by the <piAttribList> parameter of wglCreatePbufferARB:

      WGL_PBUFFER_LARGEST_ARB              0x2033

    Accepted by the <iAttribute> parameter of wglQueryPbufferARB:

      WGL_PBUFFER_WIDTH_ARB                0x2034
      WGL_PBUFFER_HEIGHT_ARB               0x2035
      WGL_PBUFFER_LOST_ARB                 0x2036
```

Additions to Chapter 2 of the 1.2 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.2 Specification (Rasterization)

None

Additions to Chapter 4 of the 1.2 Specification (Per-Fragment Operations and the Frame buffer)

None

Additions to Chapter 5 of the 1.2 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

None

Additions to the GLX Specification

This specification is written for WGL.

GLX Protocol

This specification is written for WGL.

Additions to the WGL Specification

A pixel buffer (pbuffer) can be created with wglCreatePbufferARB which returns a handle associated with the pbuffer.

```
      HPBUFFERARB wglCreatePbufferARB(HDC hDC,
                                      int iPixelFormat,
                                      int iWidth,
                                      int iHeight,
                                      const int *piAttribList);
```

<hDC> specifies a device context for the device on which the pbuffer is created. <iPixelFormat> specifies a non-generic pixel format descriptor index. Support for pbuffers may be restricted to specific pixel formats. Use wglGetPixelFormatAttribivEXT or wglGetPixelFormatAttribfvEXT to query the WGL_DRAW_TO_PBUFFER_ARB attribute to determine which pixel formats support the creation of pbuffers.

```
    <iWidth> and <iHeight> specify the pixel width and height of the
    rectangular pbuffer.

    <piAttribList> is a list of attributes {type, value} pairs
    containing integer attribute values.  All of the attributes in the
    <piAttribList> are followed by the corresponding required value.
    The list is terminated with a value of 0.

    The following attributes are supported by wglCreatePbufferARB:

      WGL_PBUFFER_LARGEST_ARB     If this attribute is set to a
                                  non-zero value, the largest
                                  available pbuffer is allocated
                                  when the allocation of the pbuffer
                                  would otherwise fail due to
                                  insufficient resources.  The width
                                  or height of the allocated pbuffer
                                  never exceeds <iWidth> and <iHeight>,
                                  respectively.  Use wglQueryPbufferARB
                                  to retrieve the dimensions of the
                                  allocated pbuffer.

    The resulting pbuffer will contain color buffers and ancillary
    buffers as specified by <iPixelFormat>.  Note that pbuffers use
    framebuffer resources so applications should consider deallocating
    them when they are not in use.

    It is possible to create a pbuffer with back buffers and to swap
    the front and back buffers by calling wglSwapLayerBuffers.  The
    contents of the back buffers after the swap depends on the
    <iPixelFormat>.  (Pbuffers are the same as windows in this respect.)

    When wglCreatePbufferARB fails to create a pbuffer, NULL is
    returned.  To get extended error information, call GetLastError.
    Possible errors are as follows:

      ERROR_INVALID_PIXEL_FORMAT     Pixel format is not valid.

      ERROR_NO_SYSTEM_RESOURCES      Insufficient resources exist.

      ERROR_INVALID_DATA             <iWidth> or <iHeight> is negative
                                     or zero.

      ERROR_INVALID_DATA             <piAttribList> is not a valid
                                     attribute.

    To create a device context for the pbuffer, call

      HDC wglGetPbufferDCARB(HPBUFFERARB hPbuffer);

    where <hPbuffer> is a handle returned from a previous call to
    wglCreatePbufferARB.  A device context is returned by
    wglGetPbufferDCARB which can be used to associate a rendering
    context with the pbuffer.  Any rendering context created with
    a wglCreateContext that is "compatible" with the <iPixelFormat> may
    be used to render into the pbuffer. (See the description of
    wglCreateContext, wglMakeCurrent, and wglMakeCurrentReadEXT for a
    definition of "compatible".)

    When wglGetPbufferDCARB fails, NULL is returned.  To get extended
    error information, call GetLastError. Possible errors are as
    follows:

      ERROR_INVALID_HANDLE           <hPbuffer> is not a valid handle.

    To release a device context obtained from a previous call to
    wglGetPbufferDCARB, call

        int wglReleasePbufferDCARB(HPBUFFERARB hPbuffer,
                                   HDC hDC);

    If the return value is a value of 1, the device context was released.
    If the device context was not released, the return value is 0.  To
    get extended error information, call GetLastError. Possible errors
    are as follows:

      ERROR_INVALID_HANDLE           <hPbuffer> is not a valid handle.
      ERROR_DC_NOT_FOUND             <hDC> is not a valid DC.

    A pbuffer is destroyed by calling

      BOOL wglDestroyPbufferARB(HPBUFFERARB hPbuffer);

    The pbuffer is destroyed once it is no longer current to any
    rendering context.  When a pbuffer is destroyed, any memory
    resources that are attached to it are freed and its handle is no
    longer valid.

    If wglDestroyPbufferARB fails, FALSE is returned.  To get extended
    error information, call GetLastError. Possible errors are as
    follows:

      ERROR_INVALID_HANDLE           <hPbuffer> is not a valid handle.

    To query the maximum width, height, or number of pixels in any
    given pbuffer for a specific pixel format, use
    wglGetPixelFormatAttribivEXT or wglGetPixelFormatAttribfvEXT with
    <attribute> set to one of WGL_MAX_PBUFFER_WIDTH_ARB,
    WGL_MAX_PBUFFER_HEIGHT_ARB, or WGL_MAX_PBUFFER_PIXELS_ARB.

    WGL_MAX_PBUFFER_WIDTH_ARB and WGL_MAX_PBUFFER_HEIGHT_ARB indicate
    the maximum width and height that can be passed into
    wglCreatePbufferARB and WGL_MAX_PBUFFER_PIXELS_ARB indicates the
    maximum number of pixels (width x height) for a pbuffer.  Note
    that an implementation may return a value for
    WGL_MAX_PBUFFER_PIXELS_ARB that is less than the maximum width
    times the maximum height.  Also, the value for
    WGL_MAX_PBUFFER_PIXELS_ARB is static and assumes that no other
    pbuffers are contending for the framebuffer memory.  Thus it may
    not be possible to allocate a pbuffer of the size given by
    WGL_MAX_PBUFFER_PIXELS_ARB.
```

To query an attribute associated with a specific pbuffer, call

```
  BOOL wglQueryPbufferARB(HPBUFFERARB hPbuffer,
                          int iAttribute,
                          int *piValue);
```

with <hPbuffer> set to a previously returned pbuffer handle. <iAttribute> must be set to one of WGL_PBUFFER_WIDTH_ARB, WGL_PBUFFER_HEIGHT_ARB, or WGL_PBUFFER_LOST_ARB.

The WGL_PBUFFER_LOST_ARB query can be used to determine if the pixel buffer memory was lost due to a display mode change. A value of TRUE is returned in <iAttribute> if the display mode change lost the memory for the pixel buffer. It is not an error to render to a pixel buffer in this state, but the effect of rendering to it is the same as if the pixel buffer was destroyed: the context state will be updated, but the values of the returned pixels are undefined. The pixel buffer must be destroyed and recreated if the pixel buffer memory has been lost. A value of FALSE is returned to indicate that the contents of the pixel buffer are unaffected by the display mode change.

If wglQueryPbufferARB fails, FALSE is returned. To get extended error information, call GetLastError. Possible errors are as follows:

```
  ERROR_INVALID_HANDLE      <hPbuffer> is not a valid handle.
  ERROR_INVALID_DATA        <iAttribute> is not a valid attribute.
```

Dependencies on WGL_ARB_pixel_format

The WGL_ARB_pixel_format extension must be used to determine a pixel format that can be used to create the pixel buffer.

Dependencies on WGL_ARB_extensions_string

Because there is no way to extend wgl, these calls are defined in the ICD and can be called by obtaining the address with wglGetProcAddress. Because this extension is a WGL extension, it is not included in the GL_EXTENSIONS string. Its existence can be determined with the WGL_ARB_extensions_string extension.

New State

None

New Implementation Dependent State

None

Conformance Testing

All of the current conformance tests can be run on a pixel buffer to validate its conformance. The only change to the conformance tests would be to create a context for the pixel buffer.

Revision History

```
    12/16/1999  0.1
        - First ARB draft based on the EXT specification.

    02/28/2000  0.2
        - Added a query for a damaged pixel buffer due to a display
          mode change.

    03/15/2000  0.3
        - Changed the lost definition of a pixel buffer.
        - Removed the OPTIMAL size queries.
        - Added a dependency on WGL_ARB_pixel_format.

    03/22/2000  1.0
        - Changed "mode change" to "display mode change".
        - Added the condition that the resources associated with a
          pbuffer may be lost due to a display mode change.
        - Fixed issue 1 to address the OPTIMUM values.
        - Added the declaration of HPBUFFERARB in the Procedures and
          Functions section.
        - Changed the wording of "undamaged" to "unaffected"
        - Approved by ARB: 10-0-0.
```

Name

    WGL_ARB_pixel_format

Name Strings

    WGL_ARB_pixel_format

Status

    Complete. Approved by ARB on 3/15/2000.

Version

    Last Modified Date: March 22, 2000
    Author Revision: 1.0

Number

    ARB Extension #9

Dependencies

    WGL_ARB_extensions_string is required.

Overview

    This extension adds functions to query pixel format attributes and
    to choose from the list of supported pixel formats.

    These functions treat pixel formats as opaque types: attributes are
    specified by name rather than by accessing them directly as fields
    in a structure. Thus the list of attributes can be easily extended.

    Attribute names are defined which correspond to all of the values in
    the PIXELFORMATDESCRIPTOR and LAYERPLANEDESCRIPTOR data structures.
    Additionally this interface allows pixel formats to be supported
    which have attributes that cannot be represented using the standard
    pixel format functions, i.e. DescribePixelFormat,
    DescribeLayerPlane, ChoosePixelFormat, SetPixelFormat, and
    GetPixelFormat.

IP Status

    No issues.

Issues and Notes

      1. No provision is made to support extended pixel format attributes in
         metafiles.
      2. Should the transparent value pixel format attribute have separate red,
         green and blue values? Yes.
      3. What data type should the transparent value be? This is no longer an
         issue since the transparent value is no longer a packed pixel value (it
         has separate r,g,b,a and index values).
      4. Should we add DONT_CARE values for some of the pixel format attributes?
         No we should just ignore attributes that aren't specified in the list

```
     passed to wglChoosePixelFormatARB.
  5. Should wglGetPixelFormatAttrib*vARB ignore the <iLayerPlane> parameter
     when the attribute specified only applies to the main planes (e.g.,
     when the attribute is set to WGL_NUMBER_OVERLAYS) or should it require
     <iLayerPlane> to be set to zero? It will just ignore the parameter.
     This allows these attributes to be queried at the same time as
     attributes of the overlay planes.
  6. Should wglGetPixelFormatAttribivARB convert floating point values to
     fixed point? No, wglChoosePixelFormatARB needs a way to accept floating
     point values. pfAttribFList accomplishes this.
  7. Should wglChoosePixelFormatARB take an <iLayerPlane> parameter?
     Typically <iLayerPlane> would be set to zero and a pixel format would
     be selected based on the attributes of the main plane, so there is no
     <iLayerPlane> parameter. This should be OK; applications won't
     typically select a pixel format on the basis of overlay attributes.
     They can always call wglGetPixelFormatAttrib*vARB to get a pixel format
     that has the desired overlay values.
  8. Application programmers must check to see if a particular extension is
     supported before using any pixel format attributes associated with the
     extension. For example, if WGL_ARB_pbuffer is not supported then it is
     an error to specify WGL_DRAW_TO_PBUFFER_ARB in the attribute list to
     wglGetPixelFormatAttrib*vARB or wglChoosePixelFormatARB.
  9. Should WGLChoosePixelFormatARB consider pixel formats at other display
     depths? It would be useful to have an argument to
     WGLChoosePixelFormatARB indicating what display depth should be used.
     However, there is no good way to implement this in the ICD since pixel
     format handles are sequential indices and the pixel format for index n
     differs depending on the display mode.
 10. Should we allow non-displayable pixel formats for pbuffers? Yes,
     although many (most?) implementations will use displayable pixel
     formats for pbuffers, this is a useful feature and the spec should
     allow for it.
 11. Should we create all new calls for pixel formats, specifically should
     we introduce SetPixelFormatARB? No, this doesn't offer any value over
     the existing SetPixelFormat call.
 12. Should we add support for triple buffering? No, triple buffering needs
     to be covered by a separate extension.
```

New Procedures and Functions

```
    BOOL wglGetPixelFormatAttribivARB(HDC hdc,
                                      int iPixelFormat,
                                      int iLayerPlane,
                                      UINT nAttributes,
                                      const int *piAttributes,
                                      int *piValues);

    BOOL wglGetPixelFormatAttribfvARB(HDC hdc,
                                      int iPixelFormat,
                                      int iLayerPlane,
                                      UINT nAttributes,
                                      const int *piAttributes,
                                      FLOAT *pfValues);
    BOOL wglChoosePixelFormatARB(HDC hdc,
                                 const int *piAttribIList,
                                 const FLOAT *pfAttribFList,
                                 UINT nMaxFormats,
                                 int *piFormats,
                                 UINT *nNumFormats);
```

New Tokens

    Accepted in the <piAttributes> parameter array of
    wglGetPixelFormatAttribivARB, and wglGetPixelFormatAttribfvARB, and
    as a type in the <piAttribIList> and <pfAttribFList> parameter
    arrays of wglChoosePixelFormatARB:

```
        WGL_NUMBER_PIXEL_FORMATS_ARB            0x2000
        WGL_DRAW_TO_WINDOW_ARB                  0x2001
        WGL_DRAW_TO_BITMAP_ARB                  0x2002
        WGL_ACCELERATION_ARB                    0x2003
        WGL_NEED_PALETTE_ARB                    0x2004
        WGL_NEED_SYSTEM_PALETTE_ARB             0x2005
        WGL_SWAP_LAYER_BUFFERS_ARB              0x2006
        WGL_SWAP_METHOD_ARB                     0x2007
        WGL_NUMBER_OVERLAYS_ARB                 0x2008
        WGL_NUMBER_UNDERLAYS_ARB                0x2009
        WGL_TRANSPARENT_ARB                     0x200A
        WGL_TRANSPARENT_RED_VALUE_ARB           0x2037
        WGL_TRANSPARENT_GREEN_VALUE_ARB         0x2038
        WGL_TRANSPARENT_BLUE_VALUE_ARB          0x2039
        WGL_TRANSPARENT_ALPHA_VALUE_ARB         0x203A
        WGL_TRANSPARENT_INDEX_VALUE_ARB         0x203B
        WGL_SHARE_DEPTH_ARB                     0x200C
        WGL_SHARE_STENCIL_ARB                   0x200D
        WGL_SHARE_ACCUM_ARB                     0x200E
        WGL_SUPPORT_GDI_ARB                     0x200F
        WGL_SUPPORT_OPENGL_ARB                  0x2010
        WGL_DOUBLE_BUFFER_ARB                   0x2011
        WGL_STEREO_ARB                          0x2012
        WGL_PIXEL_TYPE_ARB                      0x2013
        WGL_COLOR_BITS_ARB                      0x2014
        WGL_RED_BITS_ARB                        0x2015
        WGL_RED_SHIFT_ARB                       0x2016
        WGL_GREEN_BITS_ARB                      0x2017
        WGL_GREEN_SHIFT_ARB                     0x2018
        WGL_BLUE_BITS_ARB                       0x2019
        WGL_BLUE_SHIFT_ARB                      0x201A
        WGL_ALPHA_BITS_ARB                      0x201B
        WGL_ALPHA_SHIFT_ARB                     0x201C
        WGL_ACCUM_BITS_ARB                      0x201D
        WGL_ACCUM_RED_BITS_ARB                  0x201E
        WGL_ACCUM_GREEN_BITS_ARB                0x201F
        WGL_ACCUM_BLUE_BITS_ARB                 0x2020
        WGL_ACCUM_ALPHA_BITS_ARB                0x2021
        WGL_DEPTH_BITS_ARB                      0x2022
        WGL_STENCIL_BITS_ARB                    0x2023
        WGL_AUX_BUFFERS_ARB                     0x2024
```

    Accepted as a value in the <piAttribIList> and <pfAttribFList>
    parameter arrays of wglChoosePixelFormatARB, and returned in the
    <piValues> parameter array of wglGetPixelFormatAttribivARB, and the
    <pfValues> parameter array of wglGetPixelFormatAttribfvARB:

```
        WGL_NO_ACCELERATION_ARB                     0x2025
        WGL_GENERIC_ACCELERATION_ARB                0x2026
        WGL_FULL_ACCELERATION_ARB                   0x2027

        WGL_SWAP_EXCHANGE_ARB                       0x2028
        WGL_SWAP_COPY_ARB                           0x2029
        WGL_SWAP_UNDEFINED_ARB                      0x202A

        WGL_TYPE_RGBA_ARB                           0x202B
        WGL_TYPE_COLORINDEX_ARB                     0x202C
```

Additions to Chapter 2 of the 1.2 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.2 Specification (Rasterization)

None

Additions to Chapter 4 of the 1.2 Specification (Per-Fragment Operations and the Frame buffer)

None

Additions to Chapter 5 of the 1.2 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

None

Additions to the WGL Specification

Pixel Formats

WGL uses pixel format indices to refer to the pixel formats supported by a device. The standard pixel format functions DescribePixelFormat, DescribeLayerPlane, ChoosePixelFormat, SetPixelFormat, and GetPixelFormat specify pixel format attributes using the PIXELFORMATDESCRIPTOR and LAYERPLANEDESCRIPTOR data structures.

An additional set of functions may be used to query and specify pixel format attributes by name.

Querying Pixel Format Attributes

The following two functions can be used to query pixel format attributes by specifying a list of attributes to be queried and providing a buffer in which to receive the results from the query.

These functions can be used to query the attributes of both the main plane and layer planes of a given pixel format.

```
BOOL wglGetPixelFormatAttribivARB(HDC hdc,
                                  int iPixelFormat,
                                  int iLayerPlane,
                                  UINT nAttributes,
                                  const int *piAttributes,
                                  int *piValues);
```

<hdc> specifies the device context on which the pixel format is supported.

<iPixelFormat> is an index that specifies the pixel format. The pixel formats that a device context supports are identified by positive one-based integer indexes.

<iLayerPlane> specifies which plane is being queried. Positive values of <iLayerPlane> identify overlay planes, where 1 is the first overlay plane over the main plane, 2 is the second overlay plane over the first overlay plane, and so on. Negative values identify underlay planes, where -1 is the first underlay plane under the main plane, -2 is the second underlay plane under the first underlay plane and so on. Use zero for the main plane.

<nAttributes> number of attributes being queried.

<piAttributes> list containing an array of pixel format attribute identifiers which specify the attributes to be queried. The following values are accepted:

WGL_NUMBER_PIXEL_FORMATS_ARB
The number of pixel formats for the device context. The <iLayerPlane> and <iPixelFormat> parameters are ignored if this attribute is specified.

WGL_DRAW_TO_WINDOW_ARB
True if the pixel format can be used with a window. The <iLayerPlane> parameter is ignored if this attribute is specified.

WGL_DRAW_TO_BITMAP_ARB
True if the pixel format can be used with a memory bitmap. The <iLayerPlane> parameter is ignored if this attribute is specified.

WGL_ACCELERATION_ARB
Indicates whether the pixel format is supported by the driver. If this is set to WGL_NO_ACCELERATION_ARB then only the software renderer supports this pixel format; if this is set to WGL_GENERIC_ACCELERATION_ARB then the pixel format is supported by an MCD driver; if this is set to WGL_FULL_ACCELERATION_ARB then the pixel format is supported by an ICD driver.

```
        WGL_NEED_PALETTE_ARB
        A logical palette is required to achieve the best results for
        this pixel format. The <iLayerPlane> parameter is ignored if
        this attribute is specified.

        WGL_NEED_SYSTEM_PALETTE_ARB
        The hardware supports one hardware palette in 256-color mode
        only. The <iLayerPlane> parameter is ignored if this attribute
        is specified.

        WGL_SWAP_LAYER_BUFFERS_ARB
        True if the pixel format supports swapping layer planes
        independently of the main planes. If the pixel format does not
        support a back buffer then this is set to FALSE. The
        <iLayerPlane> parameter is ignored if this attribute is
        specified.

        WGL_SWAP_METHOD_ARB
        If the pixel format supports a back buffer, then this indicates
        how they are swapped. If this attribute is set to
        WGL_SWAP_EXCHANGE_ARB then swapping exchanges the front and back
        buffer contents; if it is set to WGL_SWAP_COPY_ARB then swapping
        copies the back buffer contents to the front buffer; if it is
        set to WGL_SWAP_UNDEFINED_ARB then the back buffer contents are
        copied to the front buffer but the back buffer contents are
        undefined after the operation. If the pixel format does not
        support a back buffer then this parameter is set to
        WGL_SWAP_UNDEFINED_ARB. The <iLayerPlane> parameter is ignored
        if this attribute is specified.

        WGL_NUMBER_OVERLAYS_ARB
        The number of overlay planes. The <iLayerPlane> parameter is
        ignored if this attribute is specified.

        WGL_NUMBER_UNDERLAYS_ARB
        The number of underlay planes. The <iLayerPlane> parameter is
        ignored if this attribute is specified.

        WGL_TRANSPARENT_ARB
        True if transparency is supported.

        WGL_TRANSPARENT_RED_VALUE_ARB
        Specifies the transparent red color value. Typically this value
        is the same for all layer planes. This value is undefined if
        transparency is not supported.

        WGL_TRANSPARENT_GREEN_VALUE_ARB
        Specifies the transparent green value. Typically this value is
        the same for all layer planes. This value is undefined if
        transparency is not supported.

        WGL_TRANSPARENT_BLUE_VALUE_ARB
        Specifies the transparent blue color value. Typically this value
        is the same for all layer planes. This value is undefined if
        transparency is not supported.
        WGL_TRANSPARENT_ALPHA_VALUE_ARB
        Specifies the transparent alpha value. This is reserved for
        future use.

        WGL_TRANSPARENT_INDEX_VALUE_ARB
        Specifies the transparent color index value. Typically this
        value is the same for all layer planes. This value is undefined
        if transparency is not supported.

        WGL_SHARE_DEPTH_ARB
        True if the layer plane shares the depth buffer with the main
        planes. If <iLayerPlane> is zero, this is always true.

        WGL_SHARE_STENCIL_ARB
        True if the layer plane shares the stencil buffer with the main
        planes. If <iLayerPlane> is zero, this is always true.

        WGL_SHARE_ACCUM_ARB
        True if the layer plane shares the accumulation buffer with the
        main planes. If <iLayerPlane> is zero, this is always true.

        WGL_SUPPORT_GDI_ARB
        True if GDI rendering is supported.

        WGL_SUPPORT_OPENGL_ARB
        True if OpenGL is supported.

        WGL_DOUBLE_BUFFER_ARB
        True if the color buffer has back/front pairs.

        WGL_STEREO_ARB
        True if the color buffer has left/right pairs.

        WGL_PIXEL_TYPE_ARB
        The type of pixel data. This can be set to WGL_TYPE_RGBA_ARB or
        WGL_TYPE_COLORINDEX_ARB.

        WGL_COLOR_BITS_ARB
        The number of color bitplanes in each color buffer. For RGBA
        pixel types, it is the size of the color buffer, excluding the
        alpha bitplanes. For color-index pixels, it is the size of the
        color index buffer.

        WGL_RED_BITS_ARB
        The number of red bitplanes in each RGBA color buffer.

        WGL_RED_SHIFT_ARB
        The shift count for red bitplanes in each RGBA color buffer.

        WGL_GREEN_BITS_ARB
        The number of green bitplanes in each RGBA color buffer.

        WGL_GREEN_SHIFT_ARB
        The shift count for green bitplanes in each RGBA color buffer.

        WGL_BLUE_BITS_ARB
        The number of blue bitplanes in each RGBA color buffer.
        WGL_BLUE_SHIFT_ARB
        The shift count for blue bitplanes in each RGBA color buffer.

        WGL_ALPHA_BITS_ARB
        The number of alpha bitplanes in each RGBA color buffer.

        WGL_ALPHA_SHIFT_ARB
        The shift count for alpha bitplanes in each RGBA color buffer.

        WGL_ACCUM_BITS_ARB
        The total number of bitplanes in the accumulation buffer.

        WGL_ACCUM_RED_BITS_ARB
        The number of red bitplanes in the accumulation buffer.

        WGL_ACCUM_GREEN_BITS_ARB
        The number of green bitplanes in the accumulation buffer.

        WGL_ACCUM_BLUE_BITS_ARB
        The number of blue bitplanes in the accumulation buffer.

        WGL_ACCUM_ALPHA_BITS_ARB
        The number of alpha bitplanes in the accumulation buffer.

        WGL_DEPTH_BITS_ARB
        The depth of the depth (z-axis) buffer.

        WGL_STENCIL_BITS_ARB
        The depth of the stencil buffer.

        WGL_AUX_BUFFERS_ARB
        The number of auxiliary buffers.

    <piValues> points to a buffer into which the results of the query
    will be placed. Floating point attribute values are rounded to the
    nearest integer value. The caller must allocate this array and it
    must have at least <nattributes> entries.

    If the function succeeds, the return value is TRUE. If the function
    fails, the return value is FALSE. To get extended error information,
    call GetLastError.

    An error is generated if <piAttributes> contains an invalid
    attribute, if <iPixelFormat> is not a positive integer or is larger
    than the number of pixel formats, if <iLayerPlane> doesn't refer to
    an existing layer plane, or if <hdc> is invalid.

    If FALSE is returned, the contents of <piValues> are undefined.

    BOOL wglGetPixelFormatAttribfvARB(HDC hdc,
                                      int iPixelFormat,
                                      int iLayerPlane,
                                      UINT nAttributes,
                                      const int *piAttributes,
                                      FLOAT *pfValues);
```

<hdc> specifies the device context on which the pixel format is supported.

<iPixelFormat> is an index that specifies the pixel format. The pixel formats that a device context supports are identified by positive one-based integer indexes.

<iLayerPlane> specifies which plane is being queried. Positive values of <iLayerPlane> identify overlay planes, where 1 is the first overlay plane over the main plane, 2 is the second overlay plane over the first overlay plane, and so on. Negative values identify underlay planes, where -1 is the first underlay plane under the main plane, -2 is the second underlay plane under the first underlay plane and so on. Use zero for the main plane.

<nAttributes> number of attributes being queried.

<piAttributes> list containing an array of pixel format attribute identifiers which specify the attributes to be queried. The values accepted are the same as for wglGetPixelFormatAttribivARB.

<pfValues> is a pointer to a buffer into which the results of the query will be placed. Integer attribute values are converted floating point The caller must allocate this array and it must have at least at least <nAttributes> entries.

If the function succeeds, the return value is TRUE. If the function fails, the return value is FALSE. To get extended error information, call GetLastError.

An error is generated if <piAttributes> contains an invalid attribute, if <iPixelFormat> is not a positive integer or is larger than the number of pixel formats, if <iLayerPlane> doesn't refer to an existing layer plane, or if <hdc> is invalid.

If FALSE is returned, the contents of <pfValues> are undefined.

Supported Pixel Formats

The maximum index of the pixel formats which can be referenced by the standard pixel format functions is returned by a successful call to DescribePixelFormat. This may be less than the maximum index of the pixel formats which can be referenced by wglGetPixelFormatAttribivARB and wglGetPixelFormatAttribfvARB. (determined by querying WGL_NUMBER_PIXEL_FORMATS_ARB).

The pixel format of a "displayable" object (e.g. window, bitmap) is specified by passing its index to SetPixelFormat. Therefore, pixel formats which cannot be referenced by the standard pixel format functions are "non displayable".

Indices are assigned to pixel formats in the following order:

1. Accelerated pixel formats that are displayable

2. Accelerated pixel formats that are displayable and which have extended attributes

```
    3. Generic pixel formats

    4. Accelerated pixel formats that are non displayable

    ChoosePixelFormat will never select pixel formats from either group
    2 or group 4. Each pixel format in group 2 is required to appear
    identical to some pixel format in group 1 when queried by
    DescribePixelFormat. Consequently, ChoosePixelFormat will always
    select a format from group 1 when it might otherwise have selected a
    format from group 2. Pixel formats in group 4 cannot be accessed by
    ChoosePixelFormat at all.

    SetPixelFormat and DescribePixelFormat will only accept pixel
    formats from groups 1-3. If a non-displayable pixel format is
    specified to SetPixelFormat or DescribePixelFormat an error will
    result. These pixel formats are only for use with WGL extensions,
    such as WGLCreatePbufferARB.

    The following function may be used to select from among all of the
    available pixel formats (including both accelerated and generic
    formats and non-displayable formats). This function accepts
    attributes for the main planes. A list of pixel formats that match
    the specified attributes is returned with the "best" pixel formats
    at the start of the list (order is device dependent).

    BOOL wglChoosePixelFormatARB(HDC hdc,
                                 const int *piAttribIList,
                                 const FLOAT *pfAttribFList,
                                 UINT nMaxFormats,
                                 int *piFormats,
                                 UINT *nNumFormats);

    <hdc> specifies the device context.

    <piAttribIList> specifies a list of attribute {type, value} pairs
    containing integer attribute values. All the attributes in
    <piAttribIList> are followed by the corresponding desired value. The
    list is terminated with 0. If <piAttribList> is NULL then the result
    is the same as if <piAttribList> was empty.

    <pfAttribFList> specifies a list of attribute {type, value} pairs
    containing floating point attribute values. All the attributes in
    <pfAttribFList> are followed by the corresponding desired value. The
    list is terminated with 0. If <pfAttribList> is NULL then the result
    is the same as if <pfAttribList> was empty.

    <nMaxFormats> specifies the maximum number of pixel formats to be
    returned.

    <piFormats> points to an array of returned indices of the matching
    pixel formats. The best pixel formats (i.e., closest match and best
    format for the hardware) are at the head of the list. The caller
    must allocate this array and it must have at least <nMaxFormats>
    entries.
```

<nNumFormats> returns the number of matching formats. This value may be larger than <nMaxFormats>.

If the function succeeds, the return value is TRUE. If the function fails the return value is FALSE. To get extended error information, call GetLastError. If no matching formats are found then nNumFormats is set to zero and the function returns TRUE.

If FALSE is returned, the contents of <piFormats> are undefined.

wglChoosePixelFormatARB selects pixel formats to return based on the attribute values specified in <piAttribIList> and <pfAttribFList>. Some attribute values must match the pixel format value exactly when the attribute is specified while others specify a minimum criteria, meaning that the pixel format value must meet or exceed the specified value. See the table below for details.

Attribute	Type	Match Criteria
WGL_DRAW_TO_WINDOW_ARB	boolean	exact
WGL_DRAW_TO_BITMAP_ARB	boolean	exact
WGL_ACCELERATION_ARB	enum	exact
WGL_NEED_PALETTE_ARB	boolean	exact
WGL_NEED_SYSTEM_PALETTE_ARB	boolean	exact
WGL_SWAP_LAYER_BUFFERS_ARB	boolean	exact
WGL_SWAP_METHOD_ARB	enum	exact
WGL_NUMBER_OVERLAYS_ARB	integer	minimum
WGL_NUMBER_UNDERLAYS_ARB	integer	minimum
WGL_SHARE_DEPTH_ARB	boolean	exact
WGL_SHARE_STENCIL_ARB	boolean	exact
WGL_SHARE_ACCUM_ARB	boolean	exact
WGL_SUPPORT_GDI_ARB	boolean	exact
WGL_SUPPORT_OPENGL_ARB	boolean	exact
WGL_DOUBLE_BUFFER_ARB	boolean	exact
WGL_STEREO_ARB	boolean	exact
WGL_PIXEL_TYPE_ARB	enum	exact
WGL_COLOR_BITS_ARB	integer	minimum
WGL_RED_BITS_ARB	integer	minimum
WGL_GREEN_BITS_ARB	integer	minimum
WGL_BLUE_BITS_ARB	integer	minimum
WGL_ALPHA_BITS_ARB	integer	minimum
WGL_ACCUM_BITS_ARB	integer	minimum
WGL_ACCUM_RED_BITS_ARB	integer	minimum
WGL_ACCUM_GREEN_BITS_ARB	integer	minimum
WGL_ACCUM_BLUE_BITS_ARB	integer	minimum
WGL_ACCUM_ALPHA_BITS_ARB	integer	minimum
WGL_DEPTH_BITS_ARB	integer	minimum
WGL_STENCIL_BITS_ARB	integer	minimum
WGL_AUX_BUFFERS_ARB	integer	minimum

All attributes except WGL_NUMBER_OVERLAYS_ARB, WGL_NUMBER_UNDERLAYS_ARB, WGL_SHARE_DEPTH_ARB, WGL_SHARE_STENCIL_ARB, and WGL_SHARE_ACCUM_ARB apply to the main planes and not to any layer planes. If WGL_SHARE_DEPTH_ARB, WGL_SHARE_STENCIL_ARB, and WGL_SHARE_ACCUM_ARB are specified in either <piAttribList> or <pfAttribList>, then a pixel format will only be selected if it has no overlays or underlays or if all of its overlays and underlays match the specified value. Applications that need to find a pixel format that supports a layer plane with other buffer attributes (such as WGL_SUPPORT_OPENGL_ARB set to TRUE), must go through the list that is returned and call wglGetPixelFormatAttrib*vARB to find one with the appropriate attributes.

Attributes that are specified in neither <piAttribIList> nor <pfAttribFList> are ignored (i.e., they are not looked at during the selection process). In addition the following attributes are always ignored, even if specified: WGL_NUMBER_PIXEL_FORMATS_ARB, WGL_RED_SHIFT_ARB, WGL_GREEN_SHIFT_ARB, WGL_BLUE_SHIFT_ARB, WGL_ALPHA_SHIFT_ARB, WGL_TRANSPARENT_ARB, WGL_TRANSPARENT_RED_VALUE_ARB,WGL_TRANSPARENT_GREEN_VALUE_ARB, WGL_TRANSPARENT_BLUE_VALUE_ARB, WGL_TRANSPARENT_ALPHA_VALUE_ARB, and WGL_TRANSPARENT_INDEX_ARB.

If both <piAttribIList> and <pfAttribFList> are NULL or empty then all pixel formats for this device are returned.

An error is generated if <piAttribIList> or <pfAttribFList> contain an invalid attribute or if <hdc> is invalid.

Although it is not an error, wglChoosePixelFormat and wglChoosePixelFormatARB should not be used together. It is not necessary to change existing OpenGL programs but application writers should use wglChoosePixelFormatARB whenever possible. New pixel format attributes introduced by extensions (such as the number of multisample buffers) will only be known to the new calls, wglChoosePixelFormatARB and wglGetPixelFormatAttrib*vARB..

New State

None

New Implementation Dependent State

None

Dependencies on WGL_ARB_extensions_string

Because there is no way to extend WGL, these calls are defined in the ICD and can be called by obtaining the address with wglGetProcAddress. Because this extension is a WGL extension, it is not included in the extension string returned by glGetString. Its existence can be determined with the WGL_ARB_extensions_string extension.

Revision History

Changes from EXT_pixel_format:

* Added WGL prefix to name to avoid possible name collisions
* EXT suffix changed to ARB
* Updated to new template, adding contact, status and revision sections
* Version is no longer an RCS version
* Attribute list passed to wglGetPixelFormatAttrib*v is type const
* Separate red,green,blue,alpha and index transparent values

```
    * WGL_SWAP_LAYER_BUFFERS and WGL_SWAP_METHOD values defined for single
      buffered pixel formats
    * Array of return values for wglGetPixelFormatAttrib*v and
      wglChoosePixelFormatARB is undefined if function fails
    * Error returned if iPixelFormat is zero or negative in
      wglGetPixelFormat*v
    * Under "Supported Pixel Formats", indicate that SetPixelFormat and
      DescribePixelFormat do not accept non displayable pixel formats.
      Passing one in results in an error
    * If either piAttribIList of pfAttribFList are NULL when
      wglChoosePixelFormatARB is called then it is as if they were empty
    * Clarify that wglChoosePixelFormatARB returns TRUE even if no matching
      formats found
    * wglChoosePixelFormatARB will only match an overlay attribute (eg,
      WGL_SHARE_DEPTH_ARB) if there are no overlay planes or if all
      overlay/underlay plane attributes match the specified criteria
    * Be careful about using term hardware (change to pixel format where
      appropriate)
    * wglChoosePixelFormatARB now ignores the following attributes (in
      addition to WGL_NUMBER_PIXEL_FORMATS_ARB): WGL_*_SHIFT_ARB,
      WGL_TRANSPARENT_ARB, WGL_TRANSPARENT_*_VALUE_ARB.
    * Clarify that new pixel format attributes (eg, attributes introduced by
      extensions such as multisampling) are only known to the new pixel
      format calls, wglChoosePixelFormatARB and wglGetPixelFormat*vARB.
    * Add dependency on WGL_ARB_extensions_string
```

Name

ARB_render_texture

Name Strings

WGL_ARB_render_texture

Status

Complete. Approved by ARB on June 13, 2001

Version

Last Modified Date: July 16, 2001

Number

ARB Extension #20

Dependencies

OpenGL 1.1 is required.
WGL_ARB_extension_string is required.
WGL_ARB_pixel_format is required.
WGL_ARB_pbuffer is required.
WGL_ARB_make_current_read affects the definition of this extension.
GL_ARB_texture_cube_map affects the definition of this extension
The extension is written against the OpenGL 1.2.1 Specification.

Overview

This extension allows a color buffer to be used for both rendering and texturing. When a color buffer is bound to a texture target it cannot be rendered to. Once it has been released from the texture it can be rendered to once again.

This extension may provide a performance boost and reduce memory requirements on architectures that support rendering to the same memory where textures reside and in the same memory format and layout required by texturing. The functionality is similar to CopyTexImage1D and CopyTexImage2D. However, some changes were made to make it easier to avoid copying data:

- Only color buffers of a pbuffer can be bound as a texture. It is not possible to use the color buffer of a window as a texture.

- The texture internal format is determined when the color buffer is associated with the texture, guaranteeing that the color buffer format is equivalent to the texture internal format.

- When a color buffer of a pbuffer is being used as a texture, the pbuffer can not be used for rendering; this makes it easier for implementations to avoid a copy of the image since the semantics of the pointer swap are clear.

- The application must release the color buffer from the texture before it can render to the pbuffer again. When the color buffer is bound as a texture, draw and read operations on the pbuffer are undefined.

- A mipmap attribute can be set, in which case memory will be allocated up front for mipmaps. The application can render the mipmap images or, if SGIS_generate_mipmap is supported, they can be automatically generated when the color buffer is bound as a texture.

- A texture target is associated with the pbuffer, so that cubemap images can be rendered into a single color buffer.

Note that this extension may be used in conjunction with other extensions to associate video images/buffers to pbuffers. Once the video image is associated with a pbuffer it can be used as a texture. Also, if SGIX_generate_mipmap is supported, it is possible to create a complete set of mipmap images from a single color buffer.

IP Status

There are no known IP issues.

Issues

1. Should we support 3D textures? What about 1D textures?

   3D textures - No. This adds a lot of implementation burden without having a good usage model.

   1D textures - Yes. Just a special case of 2D texture.

2. Should we allow a portion of the color buffer to be used as a texture? No, if a different size texture is needed the application can just create another pbuffer.

3. Do we need the MIPMAP_TEXTURE attribute?

   Yes this is good to have since some architectures may require all or some of the mipmaps to be stored together in memory.

4. Should we require power of 2 textures?

   Yes, we will allow an implementation to fail if the texture size is not a power of 2. This restriction can be relaxed later by the exension that allows non-power of 2 texture.

5. Should the render texture attributes be per color buffer or per drawable?

   There really isn't a mechanism for associating attributes with the color buffer. Also, allowing different render texture attributes for each color buffer makes the extension more difficult to implement without providing a very useful tool for applications.

```
 6. What should happen if the color buffer is used for rendering before it
    is released from the texture?

    There are three reasonable options: generate an error, create another
    buffer or have the rendering results be undefined.  Since this is an
    error condition, and not a useful feature, we should pick the option
    that is easiest to implement.  For now, we choose to have the rendering
    results be undefined--the rendering commands will be processed and the
    context will be updated but the pbuffer may or may  not be updated.
    Note that the pbuffer that contains the color buffer can be bound to a
    different context, so the invalid state must be stored with the pbuffer,
    not the context.. (Also the texture object that contains the
    color buffer's image may be released from the current context).

 7. Should the new pbuffer attributes be available through GL queries?

    No, like other pbuffer attributes you need to query them through the
    window system extension. This extension does not make any changes to
    OpenGL.

 8. Should we allow a subset of mipmaps to be defined?

    No.

 9. What happens when a pbuffer is bound as a texture and then a mode
    change occurs and the pbuffer is lost?

    The texture is not lost in this case. OpenGL doesn't have the notion
    of volatile textures and this extension should not introduce them.
    (It may be an interesting additional extension). When a color buffer
    is bound to a texture, it must be saved and restored by the driver,
    whenever texture memory is lost (even on a windows mode change).

10. Should there be any restrictions on the texture operations that
    can be performed on a color buffer?

    Yes. We allow TexSubImage and CopyTexSubImage calls but disallow
    TexImage and CopyTexImage calls. When a TexImage or CopyTexImage call
    is made then the color buffer is released back to the pbuffer and
    new memory is allocated for the texture. No mixing and matching of
    images is allowed. In other words, it is not possible to render a
    non-mipmapped image to a pbuffer, bind it to a texture and then
    call TexImage2D to create the other mipmap levels. Modifying any
    mipmap level via TexImage or CopyTexImage will cause the color
    buffer to be released back to the pbuffer, even if that level
    was not defined by the color buffer.

    Also, if DeleteTextures is called on the texture target, then the
    color buffer that is bound to the texture target is released back
    to the pbuffer.

    The implicit release of the color buffer is intended to work just
    like an explicit release - i.e. the color buffer is available for
    rendering without the app having to call ReleaseTexImage.
```

11. When the color buffer is released from the texture (back to the pbuffer) should the contents be preserved?

    No, this may prove difficult to implement on some architectures.

12. Should releasing the color buffer from the texture (back to the pbuffer) affect the scissor or viewport?

    No, since releasing the color buffer, does not change its size, it should not affect the scissor or viewport. The application is also responsible for updating the viewport and scissor when changing which mipmap level it is rendering to (this is similar to window resize, where the application is responsible for updating the scissor and viewport).

13. How should swap buffers work when a color buffer is bound as a texture?

    Since a color buffer (not a pbuffer) is bound to a texture, swap buffers should be a no-op. Otherwise the name of the bound buffer (FRONT, BACK) will change while it is bound. Note that swap buffers works just as for a pbuffer when the color buffer is not bound as a texture.

14. What happens when the application binds one color buffer of a pbuffer to a texture and then tries to render to another color buffer of the pbuffer?

    If any of the pbuffer's color buffers are bound to a texture, then rendering results are undefined for all color buffers of the pbuffer.

15. Should it be an error to bind a color buffer of a pbuffer to a texture, if that pbuffer is current to another thread?

    No. It is not an error to make a drawable current to two threads right now. Read and draw operations produce indeterminate results when the pbuffer is bound to a texture.

16. Should we allow color buffers of all drawables (pbuffers and windows) to be bound to textures?

    For now we just allow pbuffers. This is simpler since they are not shared with the window system and the color buffers are not part of the visible framebuffer. Also, windows can be resized at any time and handling this resize would unnecessarily complicate this extension.

17. Should we allow depth buffers to be bound as textures?

    This extension does not provide for this but it would be an interesting additional extension. When a color buffer is bound to a texture, only the color buffer is moved--ancillary buffers continue to be bound to the pbuffer.

    This extension is written such that adding depth textures should be very easy.

```
18. What happens when a color buffer is bound to a shared texture object?

    Since the color buffer is associated with the texture object itself,
    it should be shared.

19. Should we specify how this extension interacts with SGIS_generate_mipmap?

    No, since this is a potential ARB extension and SGIS_generate_mipmap
    is not. If SGIS_generate_mipmap is supported along with this extension,
    then if wglBindTexImageARB is called and both GENERATE_MIPMAP_SGIS and
    WGL_MIPMAP_TEXTURE_ARB are TRUE, then a set of mipmaps should be
    generated. This behaviour needs to be documented in the
    SGIS_generate_mipmap (or equivalent) extension.

20. Should we support borders on render textures?

    No. Although borders are part of 1.2.1, they are often not supported
    and better techniques (such as virtual textures) are starting to
    become available for paging in large textures.

21. Should wglBindTexImageARB take an attribute indicating whether
    mipmaps are defined or should this be implied from the
    WGL_MIPMAP_TEXTURE_ARB attribute of the pbuffer?

    This should be implied from the WGL_MIPMAP_TEXTURE_ARB attribute
    since GL allows controls for the applications to use only level zero
    image even if the pbuffer has been defined large enough to
    store mipmaps.

22. This extension introduces pbuffer attributes that can be modified.
    (Previously all pbuffer attributes were static and could not be
    changed.) Should we allow the non-static attributes to be set when the
    pbuffer is created or should we require the application to call
    wglSetPbufferAttribARB?

    We require the application to call wglSetPbufferAttribARB to set
    non-static Pbuffer attributes since this seems to be more consistent
    with OpenGL specification.

23. Do we need WGL_TEXTURE_FORMAT_ARB or is WGL_ALPHA_BITS_ARB enough
    to distinguish between selecting RGB vs. RGBA textures? Additionally,
    how is this parameter defined for non texture buffers.

    Resolved: In order to accommodate RGBA visuals to support RGB textures
    (i.e. ignore alpha) and to allow the specification to be extensible
    for depth textures, WGL_TEXTURE_FORMAT_ARB is required in this
    specification. This parameter is defined as WGL_NO_TEXTURE_ARB for
    non texture buffers.

24. Should luminance and Intensity texture formats be allowed?

    No. WGL doesn't support single-channel framebuffer formats. Allowing
    these formats would require a copy to reformat a RGB/RGBA framebuffer
    to a Luminance or Intensity format. If luminance framebuffer gets
    added to WGL, then this feature can be added at that time.
```

Implementation Notes

1. In order to prevent releases of a pbuffer from the texture object and to deal with implicit release followed by an explicit release, the GL implementation can keep a reference to any texture pbuffer in the texture object. When the pbuffer is released, this handle is set to NULL. Subsequent requests for releasing the texture pbuffer are ignored.

2. The implicit release of the color buffers has been specifed to work just like the explicit release so that the implementation can delete a texture object (one of the implicit free cases) without having to track whether the texture was associated with any color buffers.

Intended Usage

To define a cube map texture, single threaded case

1) Create the rendering window. Call wglChoosePixelFormatARB and find a suitable pixel format for rendering the image. Set the pixel format for the rendering window to this pixel format.

2) Create the pbuffer. Call wglChoosePixelFormatARB and find a suitable pixel format for rendering the texture. WGL_DRAW_TO_PBUFFER and WGL_BIND_TO_TEXTURE_RGB_ARB or WGL_BIND_TO_TEXTURE_RGBA_ARB must be TRUE. Create the pbuffer with this pixel format. Set the pbuffer width and height to the width and height of the level zero image. Set WGL_TEXTURE_FORMAT_ARB to be WGL_TEXTURE_RGB_ARB or WGL_TEXTURE_RGBA_ARB. Also set WGL_TEXTURE_TARGET_ARB to WGL_TEXTURE_CUBE_MAP_ARB.

3) Create a context for the pbuffer. Make the context current to the pbuffer and initialize the context's attributes.

4) Render all the cube map faces to the pbuffer. Call wglSetPbufferAttribARB to set the cube map face before rendering each face. Call glFlush.

5) Create a context for the window. Make the context current to the window and intialize the contexts attributes. Bind a texture object to the TEXTURE_CUBE_MAP_ARB target and set the texture parameters to the desired values.

6) Call wglBindTexImageARB to bind the pbuffer drawable to the cube map texture. Set <iBuffer> to WGL_FRONT or WGL_BACK depending upon which color buffer was used for rendering the cube map.

7) Render to the window using the cube map texture.

8) Call wglReleaseTexImageARB to release the color buffer of the pbuffer. Goto step 4 to generate more frames.

To define a 2D texture, single threaded case

In step 2, set the WGL_TEXTURE_TARGET_ARB to WGL_TEXTURE_2D_ARB.

```
    Since a 2D texture does not have multiple faces, in step 5
    there is no need to call wglSetPbufferAttribARB.

    In addition, if mipmaps are to be generated, the step 5 should
    be repeated multiple times with calls to wglSetPbufferAttribARB
    to set different mip levels.
```

New Procedures and Functions

```
    BOOL wglBindTexImageARB (HPBUFFERARB hPbuffer, int iBuffer)

    BOOL wglReleaseTexImageARB (HPBUFFERARB hPbuffer, int iBuffer)

    BOOL wglSetPbufferAttribARB (HPBUFFERARB hPbuffer,
        const int *piAttribList)
```

New Tokens

```
    Accepted by the <piAttributes> parameter of wglGetPixelFormatAttribivARB,
    wglGetPixelFormatAttribfvARB, and the <piAttribIList> and <pfAttribIList>
    parameters of wglChoosePixelFormatARB:

        WGL_BIND_TO_TEXTURE_RGB_ARB          0x2070
        WGL_BIND_TO_TEXTURE_RGBA_ARB         0x2071

    Accepted by the <piAttribList> parameter of wglCreatePbufferARB and
    by the <iAttribute> parameter of wglQueryPbufferARB:

        WGL_TEXTURE_FORMAT_ARB               0x2072
        WGL_TEXTURE_TARGET_ARB               0x2073
        WGL_MIPMAP_TEXTURE_ARB               0x2074

    Accepted as a value in the <piAttribList> parameter of
    wglCreatePbufferARB and returned in the value parameter of
    wglQueryPbufferARB when <iAttribute> is WGL_TEXTURE_FORMAT_ARB:

        WGL_TEXTURE_RGB_ARB                  0x2075
        WGL_TEXTURE_RGBA_ARB                 0x2076
        WGL_NO_TEXTURE_ARB                   0x2077

    Accepted as a value in the <piAttribList> parameter of
    wglCreatePbufferARB and returned in the value parameter of
    wglQueryPbufferARB when <iAttribute> is WGL_TEXTURE_TARGET_ARB:

        WGL_TEXTURE_CUBE_MAP_ARB             0x2078
        WGL_TEXTURE_1D_ARB                   0x2079
        WGL_TEXTURE_2D_ARB                   0x207A
        WGL_NO_TEXTURE_ARB                   0x2077

    Accepted by the <piAttribList> parameter of wglSetPbufferAttribARB and
    by the <iAttribute> parameter of wglQueryPbufferARB:

        WGL_MIPMAP_LEVEL_ARB                 0x207B
        WGL_CUBE_MAP_FACE_ARB                0x207C
    Accepted as a value in the <piAttribList> parameter of
    wglSetPbufferAttribARB and returned in the value parameter of
    wglQueryPbufferARB when <iAttribute> is WGL_CUBE_MAP_FACE_ARB:

        WGL_TEXTURE_CUBE_MAP_POSITIVE_X_ARB     0x207D
        WGL_TEXTURE_CUBE_MAP_NEGATIVE_X_ARB     0x207E
        WGL_TEXTURE_CUBE_MAP_POSITIVE_Y_ARB     0x207F
        WGL_TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB     0x2080
        WGL_TEXTURE_CUBE_MAP_POSITIVE_Z_ARB     0x2081
        WGL_TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB     0x2082

    Accepted by the <iBuffer> parameter of wglBindTexImageARB and
    wglReleaseTexImageARB:

        WGL_FRONT_LEFT_ARB                    0x2083
        WGL_FRONT_RIGHT_ARB                   0x2084
        WGL_BACK_LEFT_ARB                     0x2085
        WGL_BACK_RIGHT_ARB                    0x2086
        WGL_AUX0_ARB                          0x2087
        WGL_AUX1_ARB                          0x2088
        WGL_AUX2_ARB                          0x2089
        WGL_AUX3_ARB                          0x208A
        WGL_AUX4_ARB                          0x208B
        WGL_AUX5_ARB                          0x208C
        WGL_AUX6_ARB                          0x208D
        WGL_AUX7_ARB                          0x208E
        WGL_AUX8_ARB                          0x208F
        WGL_AUX9_ARB                          0x2090
```

Additions to Chapter 2 of the OpenGL 1.2.1 Specification (OpenGL Operation)

None.

Additions to Chapter 3 of the OpenGL 1.2.1 Specification (Rasterization)

None.

Additions to Chapter 4 of the OpenGL 1.2.1 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 1.2.1 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 1.2.1 Specification (State and State Requests)

None.

Additions to the WGL Specification

Add to the description of <piAttributes> in wglGetPixelFormatAttribivARB and <pfAttributes> in wglGetPixelFormatfv:

```
WGL_BIND_TO_TEXTURE_RGB_ARB
WGL_BIND_TO_TEXTURE_RGBA_ARB

True if the color buffers can be bound to a RGB/RGBA texture.
Currently only pbuffers can be bound as textures so this attribute
will only be TRUE if WGL_DRAW_TO_PBUFFER is also TRUE. It is
possible to bind a RGBA visual to a RGB texture in
which case the values in the alpha component of the visual
are ignored when the color buffer is used as a RGB texture.

Implementations may choose not to support WGL_BIND_TO_TEXTURE_RGB_ARB
for RGBA visuals.
```

Add new table entries to match criteria in description of wglChoosePixelFormatARB:

```
Attribute                       Type            Match Criteria

WGL_BIND_TO_TEXTURE_RGB_ARB     boolean            exact
WGL_BIND_TO_TEXTURE_RGBA_ARB    boolean            exact
```

Modify wglCreatePbufferARB:

```
HPBUFFERARB wglCreatePbufferARB (HDC hDC, int iPixelFormat,
    int iWidth, int iHeight, const int *piAttribList);

...

<iWidth> and <iHeight> specify the pixel width and height of the
rectangular pbuffer. If the texture format is set to
WGL_TEXTURE_RGB_ARB or WGL_TEXTURE_RGBA_ARB using
WGL_TEXTURE_FORMAT_ARB, then the pbuffer width and height
specify the size of the level zero texture image or, in the
case of a cube map texture, each level zero image.

<piAttribList> is a list of attribute {type, value} pairs containing
integer attribute values.  All of the attributes in <piAttribList>
are followed by the corresponding required value. The list is
terminated with a value of 0.

<piAttribList> may be NULL or empty in which case all attributes assume
their default values as described below.

The following attributes are supported by wglCreatePbufferARB:

WGL_TEXTURE_FORMAT_ARB

This attribute indicates the format of the texture that will be
created when a pbuffer is bound to a texture map.
It can be set to WGL_TEXTURE_RGB_ARB, WGL_TEXTURE_RGBA_ARB or
WGL_NO_TEXTURE_ARB. The default value is WGL_NO_TEXTURE_ARB.
```

```
        WGL_TEXTURE_TARGET_ARB

        This attribute indicates the target for the texture that will be
        created when the pbuffer is created with a texture format of
        WGL_TEXTURE_RGB_ARB or WGL_TEXTURE_RGBA_ARB.  This attribute can
        be set to WGL_NO_TEXTURE_ARB, WGL_TEXTURE_1D_ARB, WGL_TEXTURE_2D_ARB
        or WGL_TEXTURE_CUBE_MAP_ARB. The default value is WGL_NO_TEXTURE_ARB.

        WGL_MIPMAP_TEXTURE_ARB

        If this attribute is set to a non-zero value, and the texture format
        is set to WGL_TEXTURE_RGB_ARB or WGL_TEXTURE_RGBA_ARB, then storage
        for mipmaps will be allocated. The default value is FALSE.

        WGL_PBUFFER_LARGEST_ARB

        If this attribute is set to a non-zero value, the largest
        available pbuffer is allocated when the allocation of the pbuffer
        would otherwise fail due to insufficient resources.  The width or
        height of the allocated pbuffer never exceeds <iWidth> and <iHeight>,
        respectively. Also, if the pbuffer will be used as a texture
        (i.e., the value of the WGL_TEXTURE_TARGET_ARB attribute is
        WGL_TEXTURE_1D_ARB, WGL_TEXTURE_2D_ARB or WGL_TEXTURE_CUBE_MAP_ARB
        and texture format is WGL_TEXTURE_RGB_ARB or WGL_TEXTURE_RGBA_ARB),
        then the aspect ratio will be preserved and the new width and
        height will be valid sizes for the corresponding texture target.
        (e.g. Both the width and height will be a power of 2 if the
        implementation only supports power of 2 textures. Similarily,
        the width and height will be equal for a cube map texture).
        Use wglQueryPbufferARB to retrieve the dimensions of the
        allocated pbuffer. The default value for this attribute is FALSE.

        The resulting pbuffer will contain color buffers and ancillary
        buffers as specified by <iPixelFormat>.  Note that pbuffers use
        framebuffer resources so applications should consider deallocating
        them when they are not in use.

        It is possible to create a pbuffer with back buffers and to swap the
        front and back buffers by calling wglSwapLayerBuffers.  The
        contents of the back buffers after the swap depends on the
        <iPixelFormat>.  (Pbuffers are the same as windows in this respect.)

        The contents of the depth and stencil buffers may not be preserved
        when rendering a texture to the pbuffer and switching which image
        of the texture is rendered to (e.g., switching from rendering one
        mipmap level to rendering another).

        When wglCreatePbufferARB fails to create a pbuffer, NULL is returned.
        To get extended error information, call GetLastError. Possible
        errors are as follows:

        ERROR_INVALID_PIXEL_FORMAT  Pixel format is not valid.

        ERROR_NO_SYSTEM_RESOURCES   Insufficient resources exist.

        ERROR_INVALID_DATA          <iWidth> or <iHeight> is negative or zero.
```

```
    ERROR_INVALID_DATA          WGL_TEXTURE_TARGET_ARB attribute is
                                set to WGL_TEXTURE_CUBE_MAP_ARB, and
                                iWidth does not equal iHeight.

    ERROR_INVALID_DATA          WGL_TEXTURE_TARGET_ARB attribute is set
                                to WGL_TEXTURE_1D_ARB, and iHeight is
                                not set to one.

    ERROR_INVALID_DATA          The pixel format attribute
                                WGL_TEXTURE_FORMAT_ARB is
                                WGL_TEXTURE_RBG_ARB or WGL_TEXTURE_RGBA_ARB
                                and WGL_PBUFFER_WIDTH and/or
                                WGL_PBUFFER_HEIGHT specify an invalid
                                size for the implementation (e.g., the
                                texture size is not a power of 2).

    ERROR_INVALID_DATA          An attribute in <piAttribList> is not a
                                valid attribute.

    ERROR_INVALID_DATA          The texture format is set to
                                WGL_NO_TEXTURE_ARB and texture target
                                is set to something other than
                                WGL_NO_TEXTURE_ARB.

    ERROR_INVALID_DATA          The texture format is set to some target
                                besides WGL_NO_TEXTURE_ARB and texture
                                target is set to WGL_NO_TEXTURE_ARB.

    ....
```

Modify wglDestroyPbufferARB:

A pbuffer is destroyed by calling

```
BOOL wglDestroyPbufferARB(HPBUFFERARB hPbuffer);
```

The pbuffer is destroyed once it is no longer current to any rendering context and once all color buffers that are bound to a texture object have been released. When a pbuffer is destroyed, any memory resources that are attached to it are freed and its handle is no longer valid.

....

Add wglSetPbufferAttribARB:

To set an attribute of a pbuffer call

```
BOOL wglSetPbufferAttribARB (HPBUFFERARB hPbuffer,
                             const int *piAttribList);
```

with <hPbuffer> set to a previously returned pbuffer handle. <piAttribList> is a list of attribute {type, value} pairs containing integer values. All the attributes in <piAttribList> are followed by the corresponding desired value. The list is terminated with 0. If <piAttribList> is NULL or empty then this function is a no-op.

```
    The following values are accepted:

    WGL_MIPMAP_LEVEL_ARB

    For mipmap textures, this attribute indicates which level of the
    mipmap should be rendered. The default value is zero. If the value
    of this attribute is outside the range of supported mipmap level,
    the closest valid mipmap level is selected for rendering.

    WGL_CUBE_MAP_FACE_ARB

    For cube map textures, this attribute indicates which face of the
    cube map should be rendered; it must be set to one of

    WGL_TEXTURE_CUBE_MAP_POSITIVE_X_ARB,
    WGL_TEXTURE_CUBE_MAP_NEGATIVE_X_ARB,
    WGL_TEXTURE_CUBE_MAP_POSITIVE_Y_ARB,
    WGL_TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB,
    WGL_TEXTURE_CUBE_MAP_POSITIVE_Z_ARB,
    WGL_TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB.

    The default value is WGL_TEXTURE_CUBE_MAP_POSITIVE_X_ARB.

    If wglSetPbufferAttribARB fails, FALSE is returned.  To get extended
    error information, call GetLastError. Possible errors are as follows:

    ERROR_INVALID_HANDLE        <hPbuffer> is not a valid handle.

    ERROR_INVALID_DATA          Bad attribute specified in <piAttribList>.

    ERROR_INVALID_DATA          WGL_MIPMAP_LEVEL_ARB does not specify
                                a valid mipmap level.

    ERROR_INVALID_DATA          WGL_CUBE_MAP_IMAGE_ARB is not set to a
                                valid value.

    ....
```

Modify wglQueryPbufferARB:

```
    To query an attribute associated with a specific pbuffer, call

    BOOL wglQueryPbufferARB(HPBUFFERARB hPbuffer, int iAttribute,
                            int *piValue);

    with <hPbuffer> set to a previously returned pbuffer handle.
    <iAttribute> must be set to one of WGL_PBUFFER_WIDTH_ARB,
    WGL_PBUFFER_HEIGHT_ARB, WGL_PBUFFER_LOST_ARB, WGL_TEXTURE_TARGET_ARB,
    WGL_MIPMAP_TEXTURE_ARB, WGL_MIPMAP_LEVEL_ARB, WGL_CUBE_MAP_FACE_ARB
    or WGL_TEXTURE_FORMAT_ARB.

    The WGL_PBUFFER_LOST_ARB query can be used to determine if the pixel
    buffer memory was lost due to a display mode change.  A value of
    TRUE is returned in buffer <piValue> if the display mode change lost
    the memory for the pixel buffer. It is not an error to render to a
    pixel buffer in this state, but the effect of rendering to it is the
    same as if the pixel buffer was destroyed:  the context state will
```

be updated, but the values of the returned pixels are undefined. The pixel buffer must be destroyed and recreated if the pixel buffer memory has been lost. A value of FALSE is returned to indicate that the contents of the pixel buffer are unaffected by the display mode change.

When a color buffer of a pbuffer is bound as a texture, then the contents of that texture must be preserved until the color buffer is released. If the pbuffer is lost, any color buffers that are bound to textures will be freed when they are released back to the pbuffer by calling wglReleaseTexImage.

If wglPbufferAttribARB fails, FALSE is returned. To get extended error information, call GetLastError. Possible errors are as follows:

ERROR_INVALID_HANDLE	<hPbuffer> is not a valid handle.
ERROR_INVALID_DATA	<iAttribute> is not a valid attribute.

....

Add wglBindTexImageARB and wglReleaseTexImageARB:

The command

```
BOOL wglBindTexImageARB (HPBUFFERARB hPbuffer, int iBuffer)
```

defines a one-dimensional texture image or two-dimensional texture image or a set of two-dimensional cube map texture images. The texture image or images consist of the image data in <iBuffer> for the specified pbuffer, <hPbuffer>, and need not be copied. The texture target, the texture format and the size of the texture components are derived from attributes of pbuffer specified by <hPbuffer>.

Note that any existing images associated with the different mipmap levels of the texture object are freed (it is as if TexImage was called with an image of zero width).

The pbuffer attribute WGL_TEXTURE_FORMAT_ARB determines the base internal format of the texture. The component sizes are also determined by pbuffer attributes as shown in the table below.

Texture Component	Size
R	WGL_RED_BITS_ARB
G	WGL_GREEN_BITS_ARB
B	WGL_BLUE_BITS_ARB
A	WGL_ALPHA_BITS_ARB

Table x.x: Size of texture components

The texture targets are derived from the WGL_TEXTURE_TARGET_ARB attribute of <hPbuffer>. If the texture target for the pbuffer is WGL_TEXTURE_CUBE_MAP_ARB then <iBuffer> defines a set of cubemap images for the cube map texture objects which are bound to the current context (hereafter referred to as the current texture object). Note that when the texture target is WGL_TEXTURE_CUBE_MAP_ARB, all cube map texture targets are defined by a single call to wglBindTexImageARB. If the texture target is WGL_TEXTURE_2D_ARB, then <iBuffer> defines a 2D texture for the current 2D texture object; if the texture target is WGL_TEXTURE_1D_ARB, then <iBuffer> defines a 1D texture for the current 1D texture object.

The possible values for <iBuffer> are WGL_FRONT_LEFT_ARB, WGL_FRONT_RIGHT_ARB, WGL_BACK_LEFT_ARB, WGL_BACK_RIGHT_ARB, and WGL_AUX0_ARB through WGL_AUXn_ARB.

If <hPbuffer> is the calling thread's current drawable, wglBindTexImageARB performs an implicit glFlush.

After this function is called, the pbuffer associated with <iBuffer> is no longer available for reading or writing. Any read operation, such as glReadPixels, which reads values from any of the pbuffer's color buffers or ancillary buffers, will produce indeterminate results. In addition, any draw operation that is done to the pbuffer prior to wglReleaseTexImageARB being called, produces indeterminant results. Specifically, if the pbuffer is current to a context and thread then rendering commands will be processed and the context state will be updated but the pbuffer may or may not be written. Also, SwapBuffers is a no-op if it is called on this pbuffer.

Note that the color buffer is bound to a texture object. If the texture object is shared between contexts, then the color buffer is also shared. If a texture object is deleted before wglReleaseTexImageARB is called, then the color buffer is released and the pbuffer is made available for reading and writing.

It is not an error to call TexImage2D, TexImage1D, CopyTexImage1D or CopyTexImage2D to replace an image of a texture object that has a color buffer bound to it. However, these calls will cause the color buffer to be released back to the pbuffer and new memory will be allocated for the texture. Note that the color buffer is released even if the image that is being defined is a mipmap level that was not defined by the color buffer.

wglBindTexImageARB is ignored if there is no current rendering context.

If wglBindTexImageARB fails, FALSE is returned. To get extended error information, call GetLastError. Possible errors are as follows:

ERROR_INVALID_HANDLE	<hPbuffer> is not a valid handle.
ERROR_INVALID_DATA	<iBuffer> is not a valid value.
ERROR_INVALID_OPERATION	The pbuffer attribute WGL_TEXTURE_FORMAT_ARB is set to WGL_NO_TEXTURE_ARB.
ERROR_INVALID_OPERATION	<iBuffer> is already bound to the texture

To release a color buffer that is being used as a texture call

```
    BOOL wglReleaseTexImageARB (HPBUFFERARB hPbuffer, int iBuffer)
```

This releases the specified color buffer back to the pbuffer. The pbuffer is made available for reading and writing when it no longer has any color buffers bound as textures.

<iBuffer> must be one of WGL_FRONT_LEFT_ARB, WGL_FRONT_RIGHT_ARB, WGL_BACK_LEFT_ARB, WGL_BACK_RIGHT_ARB, or WGL_AUX0_ARB through WGL_AUXn_ARB.

The contents of the color buffer are undefined when it is first released. In particular there is no guarantee that the texture image is still present. However, the contents of other color buffers is unaffected by this call. Also, the contents of the depth, stencil and accumulation buffers are not affected by wglBindTexImageARB and wglReleaseTexImageARB.

If the specified color buffer is no longer bound to a texture (e.g., because the texture object was deleted) then this call is a noop; no error is generated.

After a color buffer is released from a texture (either explicitly by calling wglReleaseTexImageARB or implicitly by calling a routine such as TexImage2D), all texture images that were defined by the color buffer become NULL (it is as if TexImage was called with an image of zero width).

If wglReleaseTexImageARB fails, FALSE is returned. To get extended error information, call GetLastError. Possible errors are as follows:

ERROR_INVALID_HANDLE	<hPbuffer> is not a valid handle.
ERROR_INVALID_DATA	<iBuffer> is not a valid value.
ERROR_INVALID_OPERATION	The pbuffer attribute WGL_TEXTURE_FORMAT_ARB is set to WGL_NO_TEXTURE_ARB.

New State

None

Dependencies on GL_ARB_texture_cube_map

If GL_ARB_texture_cube_map is not supported then all references to WGL_TEXTURE_CUBE_MAP_POSITIVE_X_ARB, WGL_TEXTURE_CUBE_MAP_NEGATIVE_X_ARB, WGL_TEXTURE_CUBE_MAP_POSITIVE_Y_ARB, WGL_TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB, WGL_TEXTURE_CUBE_MAP_POSITIVE_Z_ARB, WGL_TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB, WGL_TEXTURE_CUBE_MAP_ARB and WGL_CUBE_MAP_FACE_ARB are deleted.

Revision History

```
    07/16/01    bpoddar     1. Added WGL_TEXTURE_CUBE_MAP_POSITIVE_*_ARB
                               enums to the new tokens section.
                            2. Added clarification on MIPMAP_LEVEL_ARB usage.
                            3. Removed 1 invalid error condition from
                               wglBindTexImage.
                            4. Changed parameter references to <parameter>.

    07/12/01    bpoddar     Fixed minor typos and added enum values.

    06/22/01    bpoddar     Minor language edits from ARB participants.

    04/09/01    bpoddar     1. Renamed WGL_TEXTURE_TYPE_ARB to
                               WGL_TEXTURE_TARGET_ARB.
                            2. Cleaned up behavior of WGL_TEXTURE_FORMAT_ARB.

    03/23/01    bpoddar     1. Updated the implementation notes section
                               with the discussion at the ARB.
                            2. Replaced ERROR_??? with specified errors
                            3. Clarified width and height selection rules
                               for WGL_PBUFFER_LARGEST.
                            4. Added policy for dealing with mip levels
                               both on Bind and Release.
                            5. Specified behavior for implicit release and
                               added comment to implementation section.
                            6. Added couple of errors to SetPbufferAttrib.

    03/06/01    bpoddar     1. Deleted references to 3D texture
                            2. Deleted references to LUMINANCE, INTENSITY
                               textures.
                            3. wglBindTexImageARB no longer provides a
                               separate mipmap attribute (issue #21).
                            4. Removed references to multiple texture
                               objects for cube maps.
                            5. Added issue # 23.
                            6. Added implementation notes section.

    12/01/00    pwomack     Updated issues list. Require non-static pbuffer
                            attributes to be set via SetPbufferAttrib (they
                            cannot be set when the pbuffer is created.)
                            The WGL_TEXTURE_TARGET_ARB attribute now takes
                            WGL_NO_TEXTURE_ARB as a value, so the app can
                            indicate that the pbuffer will never be bound
                            as a texture. If a pbuffer is created with
                            WGL_TEXTURE_TARGET_ARB set to WGL_NO_TEXTURE_ARB,
                            then an error results if an attempt is made to
                            bind it as a texture. Specified default values
                            for all attribute lists. When a color buffer is
                            bound as a texture then drawing to the pbuffer
                            gives undefined results (previously the
                            rendering was lost). When a color buffer is
                            bound as a texture, calling TexImage or
                            CopyTexImage releases the color buffer back to
                            the pbuffer.
    11/12/00    pwomack     Created. Copied from GLX extension. Added WGL
                            calls and removed all GLX-centric stuff.
```

Name

    WGL_ATI_pixel_format_float

Name Strings

    WGL_ATI_pixel_format_float

Contact

    Rob Mace, ATI Research (mace 'at' ati.com)

Status

    Complete.

Version

    Last Modified Date: December 4, 2002
    Revision: 5

Number

    278

Dependencies

    WGL_ARB_pixel_format is required.

    This extension is written against the OpenGL 1.3 Specification.

Overview

    This extension adds pixel formats with floating-point RGBA color
    components.

    The size of each float components is specified using the same
    WGL_RED_BITS_ARB, WGL_GREEN_BITS_ARB, WGL_BLUE_BITS_ARB and
    WGL_ALPHA_BITS_ARB pixel format attributes that are used for
    defining the size of fixed-point components. 32 bit floating-
    point components are in the standard IEEE float format. 16 bit
    floating-point components have 1 sign bit, 5 exponent bits,
    and 10 mantissa bits.

    In standard OpenGL RGBA color components are normally clamped to
    the range [0,1]. The color components of a float buffer are
    clamped to the limits of the range representable by their format.

Issues

    1. Should we expose a GL_FLOAT16_ATI pixel type?

       RESOLUTION: This will be exposed in a separate extension.

New Procedures and Functions

   None

New Tokens

Accepted by the <pname> parameters of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

```
    RGBA_FLOAT_MODE_ATI                     0x8820
    COLOR_CLEAR_UNCLAMPED_VALUE_ATI         0x8835
```

Accepted as a value in the <piAttribIList> and <pfAttribFList> parameter arrays of wglChoosePixelFormatARB, and returned in the <piValues> parameter array of wglGetPixelFormatAttribivARB, and the <pfValues> parameter array of wglGetPixelFormatAttribfvARB:

```
    WGL_TYPE_RGBA_FLOAT_ATI                 0x21A0
```

Additions to Chapter 2 of the OpenGL 1.3 Specification (OpenGL Operation)

Add a new Section 2.1.2, (p. 6):

2.1.2 16 Bit Floating-Point

A 16 bit floating-point number has 1 sign bit (s), 5 exponent bits (e), and 10 mantissa bits (m). The value (v) of a 16 bit floating-point number is determined by the following pseudo code:

```
  if (e != 0)
      v = (-1)^s * 2^(e-15) * 1.m  # normalized
  else if (f == 0)
      v = (-1)^s * 0               # zero
  else
      v = (-1)^s * 2^(e-14) * 0.m  # denormalized
```

It is acceptable for an implementation to treat denormalized 16 bit floating-point numbers as zero.

There are no NAN or infinity values for 16 bit floating-point.

Additions to Chapter 3 of the OpenGL 1.3 Specification (Rasterization)

Section 3.6.4, (p. 92), Add to figure 3.7 a block to "final conversion" for "RGBA float pixel data out" that says "clamp to float format range".

Section 3.6.4, (p. 102), change the first paragraph of the "Final Conversion" to:

For a color index, final conversion consists of masking the bits of the index to the left of the binary point by 2^n - 1, where n is the number of bits in an index buffer. For RGBA components the conversion is based on whether the components in the destination color buffer are fixed-point or floating-point. For fixed-point destination buffers components are clamped to [0,1]. The resulting values are converted to fixed-point according to the rules given in section 2.13.9 (Final Color Processing). For floating-point destination buffers components are clamped to the limits of the range representable by the destination format.

Additions to Chapter 4 of the OpenGL 1.3 Specification (Per-Fragment Operations and the Frame Buffer)

Chapter 4 Introduction, (p. 156), change the first line of the third paragraph to:

Color buffers consist of either unsigned integer color indices, RGB and optionally A unsigned integer values, of RGBA floating-point values.

Section 4.1.7, (p. 162), change the third paragraph of the page to:

Fixed-point destination (framebuffer) components and source (fragment) components are taken to be values represented according to the scheme given in section 2.13.9 (Final Color Processing). Floating-point destination and source components are taken as is. Constant color components are taken to be floating-point values.

Section 4.1.7, (p. 163), change the forth line of the second paragraph of "Using BlendFunc" to:

If destination color components are fixed-point, each floating-point value in this quadruplet is clamped to [0,1] and converted back to a fixed-point value in the manner described in section 2.13.9.

Section 4.1.8, (p. 165), insert after the first sentence:

Dithering has no effect if the destination color buffer components are floating-point.

Section 4.1.9, (p. 165), insert after the first sentence:

Logical operation has no effect if the destination color buffer components are floating-point.

Section 4.2.3, (p. 170), change the third paragraph to:

```
void ClearColor(float r, float g, float b, float a);
```

sets the clear value for the color buffers in RGBA mode. When clearing a fixed-point color buffer each of the specified components is clamped to [0; 1] and converted to fixed-point according to the rules of section 2.13.9. When clearing a floating-point color buffer the specified components are not clamped.

Section 4.3.2, (p. 176), change the "Conversion of RGBA values" to:

This step applies only if the GL is in RGBA mode, and then only if format is neither STENCIL INDEX nor DEPTH COMPONENT. The R, G, B, and A values form a group of elements. When reading from a fixed-point color buffer each element is taken to be a fixed-point value in [0; 1] with m bits, where m is the number of bits in the

      corresponding color component of the selected buffer (see section
      2.13.9).

    Section 4.3.2, (p. 177), change the second paragraph of the "Final
    Conversion" to:

      For a fixed-point RGBA color buffer, each component is first
      clamped to [0,1]. For floating-point RGBA color buffer, components
      are not clamped if the <type> is FLOAT, clamped to [0,1] if the
      <type> is unsigned, and clamped to [-1,1] if the <type> is signed.
      After clamping the appropriate conversion formula from table 4.7
      is applied to the component.

Additions to Chapter 5 of the OpenGL 1.3 Specification (Special Functions)

    None

Additions to Chapter 6 of the OpenGL 1.3 Specification (State and State Requests)

    None

Additions to the GLX Specification

    This specification is written for WGL.

GLX Protocol

    This specification is written for WGL.

Additions to the WGL Specification

    Modify the values accepted by WGL_PIXEL_TYPE_ARB to:

        WGL_PIXEL_TYPE_ARB
        The type of pixel data. This can be set to WGL_TYPE_RGBA_ARB,
        WGL_TYPE_RGBA_FLAOT_ARB, or WGL_TYPE_COLORINDEX_ARB.

Dependencies on WGL_ARB_pixel_format

    The WGL_ARB_pixel_format extension must be used to determine a
    pixel format with float components.

Dependencies on WGL_ARB_extensions_string

    Because this extension is a WGL extension, it is not included in
    the GL_EXTENSIONS string. Its existence can be determined with
    the WGL_ARB_extensions_string extension.

Errors

    None

New State

(table 6.19, p227) modify COLOR_CLEAR_VALUE and add COLOR_CLEAR_UNCLAMPED_VALUE:

Get Value	Type	Get Command	Initial Value	Description	Section	Attribute
COLOR_CLEAR_VALUE	C	GetFloatv	0,0,0,0	Color buffer clear value (RGBA mode) clamped to [0,1]	4.2.3	color-buffer
COLOR_CLEAR_UNCLAMPED_VALUE_ATI	4 x R	GetFloatv	0,0,0,0	Color buffer clear value (RGBA mode) unclamped	4.2.3	color-buffer

(table 6.28, p236) add the following entry:

Get Value	Type	Get Command	Minimum Value	Description	Section	Attribute
RGBA_FLOAT_MODE_ATI	B	GetBooleanv	-	True if RGBA components are floats	2.7	-

New Implementation Dependent State

None

Revision History

```
Date: 12/4/2002
Revision: 5
   - Added Section 2.1.2 16 Bit Floating-Point.

Date: 9/12/2002
Revision: 4
   - Fixed typo, CLEAR_COLOR_VALUE is really COLOR_CLEAR_VALUE.

Date: 9/11/2002
Revision: 3
   - Added enum numbers to New Tokens.
   - Added CLEAR_COLOR_UNCLAMPED_VALUE_ATI and defined behavior of
     CLEAR_COLOR_VALUE.
   - Added description of change to figure 3.7.
   - Clarified float clamping in section 3.6.4.

Date: 9/9/2002
Revision: 2
   - Changed wording of how float clamping is described in Overview.

Date: 9/6/2002
Revision: 1
   - First draft for circulation.
```

Name

    EXT_extensions_string

Name Strings

    WGL_EXT_extensions_string

Version

    $Date: 1999/04/03 08:41:12 $ $Revision: 1.3 $

Number

    168

Dependencies

    None

Overview

    This extension provides a way for applications to determine which
    WGL extensions are supported by a device. This is the foundation
    upon which other WGL extensions are built.

Issues

    Note that extensions that were previously advertised via glGetString
    (e.g., the swap interval extension) should continue to be advertised
    there so existing applications don't break. They should also be
    advertised via wglGetExtensionsStringEXT so new applications can make
    one call to find out which WGL extensions are supported.

New Procedures and Functions

```
    const char *wglGetExtensionsStringEXT(void);
```

New Tokens

    None

Additions to Chapter 2 of the 1.2 Specification (OpenGL Operation)

    None

Additions to Chapter 3 of the 1.2 Specification (Rasterization)

    None

Additions to Chapter 4 of the 1.2 Specification (Per-Fragment Operations and the Frame buffer)

    None

Additions to Chapter 5 of the 1.2 Specification (Special Functions)

    None

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

    None

Additions to the WGL Specification

    Advertising WGL Extensions

    Applications should call wglGetProcAddress to see whether or not
    wglGetExtensionsStringEXT is supported. If it is supported then it
    can be used to determine which WGL extensions are supported by the device.

```
        const char *wglGetExtensionsString(void);
```

    If the function succeeds, it returns a list of supported
    extensions to WGL. Although the contents of the string is
    implementation specific, the string will be NULL terminated and
    will contain a space-separated list of extension names. (The
    extension names themselves do not contain spaces.) If there are no
    extensions then the empty string is returned.

    If the function fails, the return value is NULL. To get extended
    error information, call GetLastError.

New State

    None

New Implementation Dependent State

    None

Name

    EXT_swap_control

Name Strings

    WGL_EXT_swap_control

Version

    Date: 1/27/1999   Revision: 1.3

Number

    172

Dependencies

    WGL_EXT_extensions_string is required.

Overview

This extension allows an application to specify a minimum periodicity of color buffer swaps, measured in video frame periods.

New Procedures and Functions

```
BOOL wglSwapIntervalEXT(int interval)

int wglGetSwapIntervalEXT(void)
```

New Tokens

None

Additions to Chapter 2 of the 1.2 GL Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.2 GL Specification (Rasterization)

None

Additions to Chapter 4 of the 1.2 GL Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the 1.2 GL Specification (Special Functions)

None

Additions to Chapter 6 of the 1.2 GL Specification (State and State Requests)

None

Additions to the WGL Specification

wglSwapIntervalEXT specifies the minimum number of video frame periods per buffer swap for the window associated with the current context. The interval takes effect when SwapBuffers or wglSwapLayerBuffer is first called subsequent to the wglSwapIntervalEXT call.

The parameter 'interval' specifies the minimum number of video frames that are displayed before a buffer swap will occur.

A video frame period is the time required by the monitor to display a full frame of video data. In the case of an interlaced monitor, this is typically the time required to display both the even and odd fields of a frame of video data. An interval set to a value of 2 means that the color buffers will be swapped at most every other video frame.

If 'interval' is set to a value of 0, buffer swaps are not synchronized to a video frame. The 'interval' value is silently clamped to the maximum implementation-dependent value supported before being stored.

The swap interval is not part of the render context state. It cannot be pushed or popped. The current swap interval for the window associated with the current context can be obtained by calling wglGetSwapIntervalEXT. The default swap interval is 1.

Because there is no way to extend wgl, this call is defined in the ICD and can be called by obtaining the address with wglGetProcAddress. Because this is not a GL extension, it is not included in the GL_EXTENSIONS string.

Errors

If the function succeeds, the return value is TRUE. If the function fails, the return value is FALSE. To get extended error information, call GetLastError.

ERROR_INVALID_DATA	The 'interval' parameter is negative.

New State

None

New Implementation Dependent State

None

Name

    WGL_NV_gpu_affinity

Name Strings

    WGL_NV_gpu_affinity

Contact

Barthold Lichtenbelt, NVIDIA (blichtenbelt 'at' nvidia.com)

Notice

Status

Completed.

Version

Last Modified Date: 11/08/2006
Author revision: 11

Number

Unassigned

Dependencies

WGL_ARB_extensions_string is required.

This extension interacts with WGL_ARB_make_current_read.

This extension interacts with WGL_ARB_pbuffer.

This extension interacts with GL_EXT_framebuffer_object

Overview

On systems with more than one GPU it is desirable to be able to select which GPU(s) in the system become the target for OpenGL rendering commands. This extension introduces the concept of a GPU affinity mask. OpenGL rendering commands are directed to the GPU(s) specified by the affinity mask. GPU affinity is immutable. Once set, it cannot be changed.

This extension also introduces the concept called affinity-DC. An affinity-DC is a device context with a GPU affinity mask embedded in it. This restricts the device context to only allow OpenGL commands to be sent to the GPU(s) in the affinity mask.

Handles for the GPUs present in a system are enumerated with the command wglEnumGpusNV. An affinity-DC is created by calling wglCreateAffinityDCNV. This function takes a list of GPU handles, which make up the affinity mask. An affinity-DC can also

```
    indirectly be created by obtaining a DC from a pBuffer handle, by
    calling wglGetPbufferDC, which in turn was created from an
    affinity-DC by calling wglCreatePbuffer.

    A context created from an affinity DC will inherit the GPU
    affinity mask from the DC. Once inherited, it cannot be changed.
    Such a context is called an affinity-context. This restricts the
    affinity-context to only allow OpenGL commands to be sent to those
    GPU(s) in its affinity mask. Once created, this context can be
    used in two ways:

      1. Make the affinity-context current to an affinity-DC. This
         will only succeed if the context's affinity mask is the same
         as the affinity mask in the DC. There is no window
         associated with an affinity DC, therefore this is a way to
         achieve off-screen rendering to an OpenGL context. This can
         either be rendering to a pBuffer, or an application created
         framebuffer object. In the former case, the affinity-mask of
         the pBuffer DC, which is obtained from a pBuffer handle,
         will be the same affinity-mask as the DC used to created the
         pBuffer handle.  In the latter case, the default framebuffer
         object will be incomplete because there is no window-system
         created framebuffer. Therefore, the application will have to
         create and bind a framebuffer object as the target for
         rendering.
      2. Make the affinity-context current to a DC obtained from a
         window. Rendering only happens to the sub rectangles(s) of
         the window that overlap the parts of the desktop that are
         displayed by the GPU(s) in the affinity mask of the context.

    Sharing OpenGL objects between affinity-contexts, by calling
    wglShareLists, will only succeed if the contexts have identical
    affinity masks.

    It is not possible to make a regular context (one without an
    affinity mask) current to an affinity-DC. This would mean a way
    for a context to inherit affinity information, which makes the
    context affinity mutable, which is counter to the premise of this
    extension.
```

New Procedures, Functions and Structures:

```
    DECLARE_HANDLE(HGPUNV);

    typedef struct _GPU_DEVICE {
      DWORD  cb;
      CHAR   DeviceName[32];
      CHAR   DeviceString[128];
      DWORD  Flags;
      RECT   rcVirtualScreen;
    } GPU_DEVICE, *PGPU_DEVICE;

    BOOL wglEnumGpusNV(UINT iGpuIndex,
                       HGPUNV *phGpu);
    BOOL wglEnumGpuDevicesNV(HGPUNV hGpu,
                             UINT iDeviceIndex,
                             PGPU_DEVICE lpGpuDevice);

    HDC wglCreateAffinityDCNV(const HGPUNV *phGpuList);

    BOOL wglEnumGpusFromAffinityDCNV(HDC hAffinityDC,
                                     UINT iGpuIndex,
                                     HGPUNV *hGpu);

    BOOL wglDeleteDCNV(HDC hdc);
```

New Tokens

New error codes set by wglShareLists, wglMakeCurrent and wglMakeContextCurrentARB:

```
    ERROR_INCOMPATIBLE_AFFINITY_MASKS_NV    0x20D0
```

New error codes set by wglMakeCurrent and wglMakeContextCurrentARB:

```
    ERROR_MISSING_AFFINITY_MASK_NV          0x20D1
```

Additions to the WGL Specification

GPU Affinity

To query handles for all GPUs in a system call:

```
        BOOL wglEnumGpusNV(UINT iGpuIndex, HGPUNV *phGPU);
```

<iGpuIndex> is an index value that specifies a GPU.

<phGPU> upon return will contain a handle for GPU number <iGpuIndex>. The first GPU will be index 0.

By looping over wglEnumGpusNV and incrementing <iGpuIndex>, starting at index 0, all GPU handles can be queried. If the function succeeds, the return value is TRUE. If the function fails, the return value is FALSE and <phGPU> will be unmodified. The function fails if <iGpuIndex> is greater or equal than the number of GPUs supported by the system.

To retrieve information about the display devices supported by a GPU call:

```
        BOOL wglEnumGpuDevicesNV(HGPUNV hGpu,
                                 UINT iDeviceIndex,
                                 PGPU_DEVICE lpGpuDevice);
```

<hGpu> is a handle to the GPU to query.

<iDeviceIndex> is an index value that specifies a display device, supported by <hGpu>, to query. The first display device will be index 0.

```
<lpGpuDevice> pointer to a GPU_DEVICE structure which will receive
information about the display device at index <iDeviceIndex>.

By looping over the function wglEnumGpuDevicesNV and incrementing
<iDeviceIndex>, starting at index 0, all display devices can be
queried. If the function succeeds, the return value is TRUE. If
the function fails, the return value is FALSE and <lpGpuDevice>
will be unmodified. The function fails if <iDeviceIndex> is
greater or equal than the number of display devices supported by
<hGpu>.

The GPU_DEVICE structure has the following members:

     typedef struct _GPU_DEVICE {
       DWORD  cb;
       CHAR   DeviceName[32];
       CHAR   DeviceString[128];
       DWORD  Flags;
       RECT   rcVirtualScreen;
     } GPU_DEVICE, *PGPU_DEVICE;

<cb> is the size of the GPU_DEVICE structure. Before calling
wglEnumGpuDevicesNV, set <cb> to the size, in bytes, of
GPU_DEVICE.

<DeviceName> is a string identifying the display device name. This
will be the same string as stored in the <DeviceName> field of the
DISPLAY_DEVICE structure, which is filled in by
EnumDisplayDevices.

<DeviceString> is a string describing the GPU for this display
device. It is the same string as stored in the <DeviceString>
field in the DISPLAY_DEVICE structure that is filled in by
EnumDisplayDevices when it describes a display adapter (and not a
monitor).

<Flags> Indicates the state of the display device. It can be a
combination of any of the following:

DISPLAY_DEVICE_ATTACHED_TO_DESKTOP      If set, the device is part
of the desktop.

DISPLAY_DEVICE_PRIMARY_DEVICE           If set, the primary
desktop is on this device. Only one device in the system can have
this set.

<rcVirtualScreen> specifies the display device rectangle, in
virtual screen coordinates. The value of <rcVirtualScreen> is
undefined if the device is not part of the desktop, i.e.
DISPLAY_DEVICE_ATTACHED_TO_DESKTOP is not set in the <Flags>
field.

The function wglEnumGpuDevicesNV can fail for a variety of
reasons. Call GetLastError to get extended error information.
Possible errors are as follows:

ERROR_INVALID_HANDLE   <hGpu> is not a valid GPU handle.
```

A new type of DC, called an affinity-DC, can be used to direct OpenGL commands to a specific GPU or set of GPUs. An affinity-DC is a device context with a GPU affinity mask embedded in it. This restricts the device context to only allow OpenGL commands to be sent to the GPU(s) in the affinity mask. An affinity-DC can be created directly, using the new function wglCreateAffinityDCNV and also indirectly by calling wglCreatePbufferARB followed by wglGetPbufferDCARB. To create an affinity-DC directly call:

```
     HDC wglCreateAffinityDCNV(const HGPUNV *phGpuList);
```

<phGpuList> is a NULL-terminated array of GPU handles to which the affinity-DC will be restricted. If an element in the list is not a GPU handle, as returned by wglEnumGpusNV, it is silently ignored.

If successful, the function returns an affinity-DC. If it fails, NULL will be returned.

To create an affinity-DC indirectly, first call wglCreatePbufferARB passing it an affinity-DC. Next, pass the handle returned by the call to wglCreatePbufferARB to wglGetPbufferDCARB to create an affinity-DC for the pBuffer. The DC returned by wglGetPbufferDCARB will have the same affinity mask as the DC used to create the pBuffer handle by calling wglCreatePbufferARB.

An affinity-DC has no window associated with it, and therefore it has no default window-system-provided framebuffer. (Note: This is terminology borrowed from EXT_framebuffer_object). A context made current to an affinity-DC will only be able to render into an application-created framebuffer object, or a pBuffer. The default window-system-framebuffer object, when bound, will be incomplete. The EXT_framebuffer_object specification defines what 'incomplete' means exactly.

A context created from an affinity-DC, by calling wglCreateContext and passing it an affinity-DC, is called an affinity-context. This context will inherit the affinity mask from the DC. This affinity-mask cannot be changed. The affinity mask restricts the affinity-context to only allow OpenGL commands to be sent to those GPU(s) in its affinity mask.

The function wglCreateAffinityDCNV can fail for a variety of reasons. Call GetLastError to get extended error information. Possible errors are as follows:

ERROR_NO_SYSTEM_RESOURCES	Insufficient resources exist to create the affinity-DC.
ERROR_INVALID_DATA	<phGpuList> is empty or contains no valid GPU handles

An affinity-context can only be made current to an affinity-DC with the same affinity-mask, otherwise wglMakeCurrent and wglMakeContextCurrentARB will fail and return FALSE. In the case

```
of wglMakeContextCurrentARB, the affinity masks of both the "read"
and "draw" DCs need to match the affinity-mask of the context.

If a context that has no affinity mask is made current to an
affinity-DC, wglMakeCurrent and wglMakeContextCurrentARB will fail
and return FALSE. In the case of wglMakeContextCurrentARB it will
fail if either the "read" or "draw" DC is an affinity-DC.

If an affinity-context is made current to a DC obtained from a
window, by calling GetDC, then rendering will only happen to the
subrectangle(s) of the window that overlap the parts of the
desktop that are displayed by the GPU(s) in the affinity-mask of
the context. Note that a DC obtained from a window does not have
an affinity mask set.

The following error codes are added to the description of
wglMakeCurrent and wglMakeContextCurrentARB:

ERROR_INCOMPATIBLE_AFFINITY_MASKS_NV    The device context(s) and
rendering context have non-matching affinity masks.

ERROR_MISSING_AFFINITY_MASK_NV          The rendering context does
not have an affinity mask set.

Sharing OpenGL objects between affinity-contexts, by calling
wglShareLists, will only succeed if the contexts have identical
affinity masks. The following error codes are added to the
description of wglShareLists:

ERROR_INCOMPATIBLE_AFFINITY_MASKS_NV    The contexts have non-
matching affinity masks.

To delete an affinity-DC call:

     BOOL wglDeleteDCNV(HDC hdc)

<hdc> Is a handle of an affinity-DC to delete.

If the function succeeds, TRUE is returned. If the function fails,
FALSE is returned. Call GetLastError to get extended error
information. Possible errors are as follows:

ERROR_INVALID_HANDLE   <hdc> is not a handle of an affinity-DC.

To retrieve a list of GPU handles that make up the affinity-mask
of an affinity-DC, call:

     BOOL wglEnumGpusFromAffinityDCNV(HDC hAffinityDC,
                                      UINT iGpuIndex,
                                      HGPUNV *phGpu);

<hAffinityDC> is a handle of the affinity-DC to query.

<iGpuIndex> is an index value of the GPU handle in the affinity
mask of <hAffinityDC> to query.
```

```
      <phGpu> upon return will contain a handle for GPU number
      <iGpuIndex>. The first GPU will be at index 0.

      By looping over wglEnumGpusFromAffinityDCNV and incrementing
      <iGpuIndex>, starting at index 0, all GPU handles associated with
      the DC can be queried. If the function succeeds, the return value
      is TRUE. If the function fails, the return value is FALSE and
      <phGPU> will be unmodified. The function fails if <iGpuIndex> is
      greater or equal than the number of GPUs associated with
      <hAffinityDC>.

      Call GetLastError to get extended error information. Possible
      errors are as follows:

      ERROR_INVALID_HANDLE   <hAffinityDC> is not a handle of an
      affinity-DC.
```

Interactions with WGL_ARB_make_current_read

```
      If the make current read extension is not supported, all language
      referring to wglMakeContextCurrentARB is deleted.
```

Interactions with WGL_ARB_pbuffer

```
      If the pbuffer extension is not supported, all language referring
      to puffers, wglGetPbuferDC and wglCreatePbuffer are deleted.
```

Interactions with GL_EXT_framebuffer_object

```
      If the framebuffer object extension is not supported, all language
      referring to framebuffer objects is deleted.
```

Usage examples

```
      // Example 1 - Normal window creation, DC setup and
      // context creation.

      PIXELFORMATDESCRIPTOR pfd;
      int   pf;
      HDC   hDC;
      HGLRC hRC;
      HWND  hWnd;

      hWnd = CreateWindow(...);
      hDC = GetDC(hWnd);

      memset(&pfd, 0, sizeof(pfd));
      pfd.nSize        = sizeof(pfd);
      pfd.nVersion     = 1;
      pfd.dwFlags      = PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL;
      pfd.iPixelType   = PFD_TYPE_RGBA;
      pfd.cColorBits   = 32;
```

```
// Note, for ease of code reading no error checking is done.
pf = ChoosePixelFormat(hDC, &pfd);
SetPixelFormat(hDC, pf, &pfd);
DescribePixelFormat(hDC, pf, sizeof(PIXELFORMATDESCRIPTOR),
                    &pfd);

hRC = wglCreateContext(hDC);
wglMakeCurrent(hDC, hRC);

// Example 2 - Offscreen rendering to one GPU using a FBO
// It is assumed that a context already has been created (and
// possibly destroyed) and was used to query the proc addresses
// of the WGL affinity related entrypoints.

#define MAX_GPU 4

PIXELFORMATDESCRIPTOR pfd;
int    pf, gpuIndex = 0;
HGPUNV hGPU[MAX_GPU];
HGPUNV GpuMask[MAX_GPU];
HDC    affDC;
HGLRC  affRC;

// Get a list of the first MAX_GPU GPUs in the system
while ((gpuIndex < MAX_GPU) && wglEnumGpusNV(gpuIndex,
&hGPU[gpuIndex])) {
      gpuIndex++;
}

// Create an affinity-DC associated with the first GPU
GpuMask[0] = hGPU[0];
GpuMask[1] = NULL;

affDC = wglCreateAffinityDCNV(GpuMask);

// Set a pixelformat on the affinity-DC
pf = ChoosePixelFormat(affDC, &pfd);
SetPixelFormat(affDC, pf, &pfd);
DescribePixelFormat(affDC, pf, sizeof(PIXELFORMATDESCRIPTOR),
&pfd);

affRC = wglCreateContext(affDC);
wglMakeCurrent(affDC, affRC);

// Make a previously created FBO current so we have something
// to render into. Since there's no window, the default system
// created FBO is incomplete.
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);

 <Now draw>
```

```
// **Example 3** - Offscreen rendering to one GPU using a pBuffer
// It is assumed that a context already has been created (and
// possibly destroyed) and was used to query the proc addresses
// of the WGL affinity and pbuffer related entrypoints.

#define MAX_GPU 4

int    gpuIndex = 0;
HGPUNV hGPU[MAX_GPU];
HGPUNV GpuMask[MAX_GPU];
HDC    affDC, pBufferAffDC;
HGLRC  affRC;

// Get a list of the first MAX_GPU GPUs in the system
while ((gpuIndex < MAX_GPU) && wglEnumGpusNV(gpuIndex,
&hGPU[gpuIndex])) {
      gpuIndex++;
}

// Create an affinity-DC associated with the first GPU
GpuMask[0] = hGPU[0];
GpuMask[1] = NULL;

affDC = wglCreateAffinityDCNV(GpuMask);

// Setup desired pixelformat attributes for the pbuffer
// including WGL_DRAW_TO_PBUFFER_ARB.
HPBUFFERARB  handle;
int          width = 512, height = 512, format = 0;
unsigned int nformats;

int attribList[] =
{
    WGL_RED_BITS_ARB,               8,
    WGL_GREEN_BITS_ARB,             8,
    WGL_BLUE_BITS_ARB,              8,
    WGL_ALPHA_BITS_ARB,             8,
    WGL_STENCIL_BITS_ARB,           0,
    WGL_DEPTH_BITS_ARB,             0,
    WGL_DRAW_TO_PBUFFER_ARB,        true,
    0,
};

wglChoosePixelFormatARB(affDC, attribList, NULL, 1,
                        &format, &nformats);

handle = wglCreatePbufferARB(affDC, format, width, height, NULL);

// pbufferAffDC will have the same affinity-mask as affDC.
pBufferAffDC = wglGetPbufferDCARB(handle);

// affRC will inherit the affinity-mask from pBufferAffDC.
affRC = wglCreateContext(pBufferAffDC);
wglMakeCurrent(pBufferAffDC, affRC);

<Now draw into the pBuffer>
```

Issues

1) Do we really need an affinity-DC, or can we do with just an affinity context?

DISCUSSION: If affinity is not part of a DC, a new function will need to be defined to create an affinity-context or set an affinity-mask for an existing context. Passing NULL as a HDC to wglMakeCurrent will then be one way to create an off-screen rendering context, where rendering will have to go to a FBO. If the HDC passed to wglMakeCurrent is one for a pBuffer, the affinity-mask in the affinity-context dictates where rendering is direct to. This might mean pBuffer resources will have to move, or alternatively, duplicated across all GPUs in a system. That is counter to the whole idea of this extension. Thus an affinity-DC is definitely needed for a pBuffer.

Thus the question reduces to, do we need an affinity-DC in order to facilitate off-screen rendering to a FBO? Having an affinity-DC has the following advantages:

a) It is consistent with making current to a pBuffer or window, that does need a DC.
b) passing NULL as a HDC to wglMakeCurrent might be filtered out by the MS layer on future OSes.
c) The driver implementation might benefit from knowing at DC creation time what the affinity-mask is, rather than at wglMakeCurrent time.

RESOLUTION: Yes.

2) Should the GPU affinity concept also apply to D3D and/or GDI commands?

DISCUSSION: It could be especially desirable to apply the affinity concept to D3D. However, D3D is sufficiently different that this extension doesn't directly apply.

RESOLUTION: That falls outside this extension.

3) Should setting a pixelformat on an affinity-DC be required?

DISCUSSION: Setting a pixelformat on an affinity-DC is not strictly necessary if the application does off-screen rendering to a FBO. However, the Microsoft layer of wglMakeCurrent requires that the pixelformats of the DC and RC passed to it match. This becomes an issue when making an affinity-context current to a DC obtained from a window. The DC has a pixelformat set by the application, and therefore the affinity-context needs to have the same pixelformat. This means the affinity-DC, that the affinity-context is created from, needs to have the same pixelformat set.

RESOLUTION: YES. Setting a pixelformat on an affinity-DC is required.

4) Is it allowed to make an affinity-context current to an affinity-DC where the mask of the context spans more GPUs than the mask in the DC?

5) Is it allowed to make an affinity-context current to an affinity-DC where the mask of the context spans less GPUs than the mask in the DC?

DISCUSSION: Issues 4 and 5 are lumped together in this discussion. For example, is this scenario something we want to support: An application wants to share objects across two contexts and have these two contexts each render to a different GPU. It can do this by creating two affinity-DCs. One has an affinity mask for the first GPU, the other for the second GPU. It also creates two affinity-contexts that both have an affinity-mask that spans both GPUs. Making one context current to the first affinity-DC will lock the context to the GPU in the mask of that affinity-DC. Make another context current to the second affinity-DC will lock that context to the second GPU. This is effectively what issue 4) is asking. . The simplest solution is to disallow these cases, and that is how the spec is currently written.

RESOLUTION: NO, we will not allow this to keep the spec simple. If necessary, these restrictions can always be lifted later.

6) What should an application do if the enum functions that return BOOL fail for another reason than they are done? For example, if they fail because they run out of memory?

RESOLUTION: An application will have to call GetLastError to find out the reason of failure.

7) The "Enum" API commands in this extension assume that the list of things being enumerated does not change dynamically. Is that reasonable?

DISCUSSION: Display devices, and possibly GPUs in the future, can be changed dynamically and/or hotplugged. Thus yes, this is a potential issue. Existing OS functionality like EnumDisplayDevices and even wglMakeCurrent will suffer from this too. In the latter case, the application could make a context current to a device that was removed from the system. A possible solution would be some sort of notification mechanism to the application. Possibly combined with being able to snapshot state first, then enumerate that snapshot. That snapshot of state might immediately become invalid, but at least the enumeration will walk a consistent list.

RESOLUTION: This is a wider issue than just this specification, and not currently addressed.

8) How do I transfer data efficiently between two affinity-contexts?

DISCUSSION: It is desired for an application to render in one context, and transfer the result of that rendering to another context. These two contexts can be on different GPUs. If they are, how does the application efficiently transfer this data? Currently OpenGL provides two mechanisms, neither of which are ideal:

1) The application can do a ReadPixels followed by a DrawPixels / TexImage call. This involves transfer through host memory, which can be slow.

2) The application can share objects among the two contexts using wglShareLists(). This will work, but is counter to the premise of this extension where each GPU has its own set of resources, not shared with another GPU.

RESOLUTION: This is a hole which needs to be addressed separately.

Revision history

None

Name

    NV_render_depth_texture

Name Strings

    WGL_NV_render_depth_texture

Notice

    Copyright NVIDIA Corporation, 2001, 2002.

Status

    Shipping, March 2002.

Version

    Last Modified Date: $Date: 2002/03/22 $
    NVIDIA Revision: $Revision: #5 $

Number

    263

Dependencies

    OpenGL 1.1 is required.

    ARB_render_texture is required.

    SGIX_depth_texture is required.

    NV_render_texture_rectangle affects the definition of this extension.

Overview

    This extension allows a depth buffer to be used for both rendering and
    texturing. It is built upon the ARB_render_texture extension; the only
    addition in this extension is the ability to use a depth buffer as a
    DEPTH_COMPONENT texture map.

Issues

    In the ARB_render_texture spec, the number and size of physical depth
    buffers in a rendered texture is left undefined. From the
    ARB_render_texture specification:

        The contents of the depth and stencil buffers may not be preserved
        when rendering a texture to the pbuffer and switching which image
        of the texture is rendered to (e.g., switching from rendering one
        mipmap level to rendering another).

That behavior is clearly unacceptable in an implementation where the rendered texture IS the depth buffer.

RESOLVED: Yes, it needs to be fixed. This extension specifies that each mipmap level and cube map face gets its own depth buffer, whose contents are preserved when switching render targets.

Should there be separate pixel format attributes for BIND_TO_TEXTURE_DEPTH and BIND_TO_TEXTURE_RECTANGLE_DEPTH? Or is a single attribute sufficient?

RESOLVED: We should support separate capabilities, as done with the other formats. See the NV_render_texture_rectangle spec for more info.

Should it be possible to have a single pbuffer support binding both color and depth buffers to textures?

RESOLVED: Yes. This means that we must provide a separate DEPTH_TEXTURE_FORMAT attribute that must be set at pbuffer creation time, since using only the TEXTURE_FORMAT attribute would allow you to create a pbuffer supporting either color or depth textures, but not both.

For double-buffered or stereo pixel formats that support binding to depth textures, how many depth buffers do you have?

RESOLVED: There is only a single depth buffer for double-buffered or stereo pixel formats. Double buffering refers only to the number of color buffers. There will be multiple depth buffers only if the pbuffer is specified to support mipmaps or cube maps.

What happens with multisample pixel formats, where the only depth buffer contains multiple samples per pixel? This issue is slightly different for rendered depth textures, since multisample pixel formats do contain "normal" color buffers in addition to the multisample buffer.

UNRESOLVED.

New Procedures and Functions

None.

New Tokens

Accepted by the <piAttributes> parameter of wglGetPixelFormatAttribivARB, wglGetPixelFormatAttribfvARB, and the <piAttribIList> and <pfAttribIList> parameters of wglChoosePixelFormatARB:

```
    WGL_BIND_TO_TEXTURE_DEPTH_NV                    0x20A3
    WGL_BIND_TO_TEXTURE_RECTANGLE_DEPTH_NV          0x20A4
```

Accepted by the <piAttribList> parameter of wglCreatePbufferARB and by the <iAttribute> parameter of wglQueryPbufferARB:

```
    WGL_DEPTH_TEXTURE_FORMAT_NV                     0x20A5
Accepted as a value in the <piAttribList> parameter of wglCreatePbufferARB
and returned in the value parameter of wglQueryPbufferARB when
<iAttribute> is WGL_DEPTH_TEXTURE_FORMAT_NV:

    WGL_TEXTURE_DEPTH_COMPONENT_NV                  0x20A6
    WGL_NO_TEXTURE_ARB                              0x2077

Accepted by the <iBuffer> parameter of wglBindTexImageARB:

    WGL_DEPTH_COMPONENT_NV                          0x20A7
```

Additions to Chapter 2 of the OpenGL 1.2.1 Specification (OpenGL Operation)

    None.

Additions to Chapter 3 of the OpenGL 1.2.1 Specification (Rasterization)

    None.

Additions to Chapter 4 of the OpenGL 1.2.1 Specification (Per-Fragment Operations and the Frame Buffer)

    None.

Additions to Chapter 5 of the OpenGL 1.2.1 Specification (Special Functions)

    None.

Additions to Chapter 6 of the OpenGL 1.2.1 Specification (State and State Requests)

    None.

Additions to the WGL Specification

First, close your eyes and pretend that a WGL specification actually existed. Maybe if we all concentrate hard enough, one will magically appear.

(Add to the description of <piAttributes> in wglGetPixelFormatAttribivARB and <pfAttributes> in wglGetPixelFormatfv:)

```
        WGL_BIND_TO_TEXTURE_DEPTH_NV
        WGL_BIND_TO_TEXTURE_RECTANGLE_DEPTH_NV
```

True if the depth buffer can be bound to a DEPTH_COMPONENT texture or texture rectangle. Currently only pbuffers can be bound as textures so this attribute will only be TRUE if WGL_DRAW_TO_PBUFFER is also TRUE.

(Add new table entries to match criteria in description of wglChoosePixelFormatARB:)

Attribute	Type	Match Criteria
WGL_BIND_TO_TEXTURE_DEPTH_NV	boolean	exact
WGL_BIND_TO_TEXTURE_RECTANGLE_DEPTH_NV	boolean	exact

(In the wglCreatePbufferARB section, modify the attribute list)

    WGL_TEXTURE_FORMAT_ARB

    This attribute indicates the base internal format of the texture that
    will be created when a color buffer of a pbuffer is bound to a texture
    map. It can be set to WGL_TEXTURE_RGB_ARB (indicating an internal
    format of RGB), WGL_TEXTURE_RGBA_ARB (indicating a base internal
    format of RGBA), or WGL_NO_TEXTURE_ARB. The default value is
    WGL_NO_TEXTURE_ARB.

    WGL_DEPTH_TEXTURE_FORMAT_NV

    This attribute indicates the base internal format of the texture that
    will be created when the depth buffer of a pbuffer is bound to a
    texture map. It can be set to WGL_TEXTURE_DEPTH_COMPONENT_NV
    (indicating an internal format of DEPTH_COMPONENT), or
    WGL_NO_TEXTURE_ARB. The default value is WGL_NO_TEXTURE_ARB.

(In the wglCreatePbufferARB section, modify the discussion of what happens to the depth/stencil/accum buffers when switching between mipmap levels or cube map faces.)

For pbuffers with a texture format of WGL_TEXTURE_RGB_ARB or WGL_TEXTURE_RGBA_ARB, there will be a separate set of color buffers for each mipmap level and cube map face in the pbuffer. Otherwise, the WGL implementation is free to share a single set of color, auxillary, and accumulation buffers between levels or faces.

For pbuffers with a depth texture format of WGL_TEXTURE_DEPTH_COMPONENT_NV, there will be a separate depth buffer for each mipmap level and cube map face. Otherwise, the WGL implementation is free to share a single depth buffer between levels or faces.

The contents of any color or depth buffer that may be shared between faces are undefined after switching between mipmap levels or cube map faces.

(In the wglCreatePbufferARB section, add to the error list)

    ERROR_INVALID_DATA   WGL_DEPTH_TEXTURE_FORMAT_NV is
                           WGL_TEXTURE_DEPTH_COMPONENT_NV,
                           WGL_TEXTURE_TARGET_ARB is
                           WGL_TEXTURE_RECTANGLE_NV, and the
                           WGL_BIND_TO_TEXTURE_RECTANGLE_DEPTH_NV
                           attribute is not set in the pixel format.

```
    ERROR_INVALID_DATA     WGL_DEPTH_TEXTURE_FORMAT_NV is
                           WGL_TEXTURE_DEPTH_COMPONENT_NV,
                           WGL_TEXTURE_TARGET_ARB is not
                           WGL_TEXTURE_RECTANGLE_NV, and the
                           WGL_BIND_TO_TEXTURE_DEPTH_NV attribute is not
                           set in the pixel format.
```

(In the wglCreatePbufferARB section, modify the error list, replacing the errors concerning texture format/target combinations with the following.)

```
    ERROR_INVALID_DATA     WGL_TEXTURE_TARGET_ARB is WGL_NO_TEXTURE_ARB
                           and either WGL_TEXTURE_FORMAT_ARB or
                           WGL_DEPTH_TEXTURE_FORMAT_NV is not
                           WGL_NO_TEXTURE_ARB.

    ERROR_INVALID_DATA     WGL_TEXTURE_TARGET_ARB is not
                           WGL_NO_TEXTURE_ARB and both
                           WGL_TEXTURE_FORMAT_ARB and
                           WGL_DEPTH_TEXTURE_FORMAT_NV are
                           WGL_NO_TEXTURE_ARB.
```

Modify wglDestroyPbufferARB:

A pbuffer is destroyed by calling

```
BOOL wglDestroyPbufferARB(HPBUFFERARB hPbuffer);
```

The pbuffer is destroyed once it is no longer current to any rendering context and once all color and depth buffers that are bound to a texture object have been released. When a pbuffer is destroyed, any memory resources that are attached to it are freed and its handle is no longer valid.

....

Modify wglBindTexImageARB:

...

The pbuffer attribute WGL_DEPTH_TEXTURE_FORMAT_NV determines the base internal format of the depth texture. The format-specific component sizes are also determined by pbuffer attributes as shown in the table below. The component sizes are dependent on the format of the texture.

Texture Component	Size	Format
D	WGL_DEPTH_BITS_ARB	DEPTH_COMPONENT

Table x.x: Size of texture components

...

The possible values for <iBuffer> are WGL_FRONT_LEFT_ARB, WGL_FRONT_RIGHT_ARB, WGL_BACK_LEFT_ARB, WGL_BACK_RIGHT_ARB, WGL_DEPTH_COMPONENT_NV, and WGL_AUX0_ARB through WGL_AUXn_ARB.

    ...

(Modify paragraphs in wglBindTexImageARB section to include language about allowing depth buffers)

Note that the color or depth buffer is bound to a texture object. If the texture object is shared between contexts, then the color or depth buffers are also shared. If a texture object is deleted before wglReleaseTexImageARB is called, then the color buffer is released and the pbuffer is made available for reading and writing.

It is not an error to call TexImage2D, TexImage1D, CopyTexImage1D or CopyTexImage2D to replace an image of a texture object that has a color or depth buffer bound to it. However, these calls will cause the color or depth buffers to be released back to the pbuffer and new memory will be allocated for the texture. Note that the color or depth buffer is released even if the image that is being defined is a mipmap level that was not defined by the color buffer.

(Modify wglReleaseTexImageARB section to include language allowing the binding of depth buffers)

To release a color or depth buffer that is being used as a texture call

```
    BOOL wglReleaseTexImageARB (HPBUFFERARB hPbuffer, int iBuffer)
```

This releases the specified color or depth buffer back to the pbuffer. The pbuffer is made available for reading and writing when it no longer has any color or depth buffers bound as textures.

<iBuffer> must be one of WGL_FRONT_LEFT_ARB, WGL_FRONT_RIGHT_ARB, WGL_BACK_LEFT_ARB, WGL_BACK_RIGHT_ARB, WGL_DEPTH_COMPONENT_NV, or WGL_AUX0_ARB through WGL_AUXn_ARB.

The contents of the color or depth buffer being released are undefined when it is first released. In particular, there is no guarantee that the texture image is still present. However, the contents of other color, depth, stencil, or accumulation buffers are unaffected when the color or depth buffer is released.

If the specified color or depth buffer is no longer bound to a texture (e.g., because the texture object was deleted) then this call is a noop; no error is generated.

After a color or depth buffer is released from a texture (either explicitly by calling wglReleaseTexImageARB or implicitly by calling a routine such as TexImage2D), all texture images that were defined by the color buffer become NULL (it is as if TexImage was called with an image of zero width).

New State

None

Dependencies on NV_render_texture_rectangle

    If NV_render_texture_rectangle is not supported, all references to texture
    rectangles and WGL_BIND_TO_TEXTURE_RECTANGLE_DEPTH_NV should be deleted.

Name

    NV_render_texture_rectangle

Name Strings

    WGL_NV_render_texture_rectangle

Notice

Status

    Shipping, March 2002.

Version

```
    Last Modified Date:  $Date: 2003/01/08 $
    NVIDIA Revision:  $Revision: #7 $
```

Number

    264

Dependencies

    OpenGL 1.1 is required.

    WGL_ARB_render_texture is required.

    GL_NV_texture_rectangle is required.

    The extension is written against the OpenGL 1.2.1 Specification.

Overview

    This extension allows a color buffer with non-power-of-two dimensions to
    be used for both rendering and texturing. It is built upon the
    ARB_render_texture extension; the only addition in this extension is the
    ability to bind a texture to a texture rectangle target, as provided
    through the NV_texture_rectangle extension.

Issues

    What is the interaction of this spec and the WGL_MIPMAP_TEXTURE_ARB
    attribute?

      RESOLVED: NV_texture_rectangle doesn't support mipmaps, so it's kind of
      stupid to allocate them. Trying will result in an error.

    Should there be separate pixel format attributes for
    BIND_TO_TEXTURE_RECTANGLE_RGB and RGBA? Or is a simple
    BIND_TO_TEXTURE_RECTANGLE attribute sufficient?

      RESOLVED: Separate capabilities. There may be pixel formats where
      rendered texture rectangles are supported, but conventional textures are

    not. If a single BIND_TO_TEXTURE_RECTANGLE attribute were used, there
    would be no cue for RGB/RGBA binding support, and the existing
    attributes would signal the ability to render to conventional textures.

    Alternately, pixel formats could be constrained so that the only
    render-texture capable formats are those that support all allowable
    targets.

Implementation Notes

None.

New Procedures and Functions

None.

New Tokens

Accepted by the <piAttributes> parameter of wglGetPixelFormatAttribivARB, wglGetPixelFormatAttribfvARB, and the <piAttribIList> and <pfAttribIList> parameters of wglChoosePixelFormatARB:

```
    WGL_BIND_TO_TEXTURE_RECTANGLE_RGB_NV           0x20A0
    WGL_BIND_TO_TEXTURE_RECTANGLE_RGBA_NV          0x20A1
```

Accepted as a value in the <piAttribList> parameter of wglCreatePbufferARB and returned in the value parameter of wglQueryPbufferARB when <iAttribute> is WGL_TEXTURE_TARGET_ARB:

```
    WGL_TEXTURE_RECTANGLE_NV                       0x20A2
```

Additions to Chapter 2 of the OpenGL 1.2.1 Specification (OpenGL Operation)

None.

Additions to Chapter 3 of the OpenGL 1.2.1 Specification (Rasterization)

None.

Additions to Chapter 4 of the OpenGL 1.2.1 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 1.2.1 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 1.2.1 Specification (State and State Requests)

None.

Additions to the WGL Specification

First, close your eyes and pretend that a WGL specification actually existed. Maybe if we all concentrate hard enough, one will magically appear.

These modifications are relative to the ARB_render_texture spec.

Add to the description of <piAttributes> in wglGetPixelFormatAttribivARB and <pfAttributes> in wglGetPixelFormatfv:

```
WGL_BIND_TO_TEXTURE_RECTANGLE_RGB_NV
WGL_BIND_TO_TEXTURE_RECTANGLE_RGBA_NV
```

True if the color buffers can be bound as RGB/RGBA textures using the texture rectangle target. Currently only pbuffers can be bound as textures so this attribute will only be TRUE if WGL_DRAW_TO_PBUFFER is also TRUE. It is possible to bind a RGBA visual to a RGB texture in which case the values in the alpha component of the visual are ignored when the color buffer is used as a RGB texture.

Add new table entries to match criteria in description of wglChoosePixelFormatARB:

Attribute	Type	Match Criteria
WGL_BIND_TO_TEXTURE_RECTANGLE_RGB_NV	boolean	exact
WGL_BIND_TO_TEXTURE_RECTANGLE_RGBA_NV	boolean	exact

Modify wglCreatePbufferARB:

The following attributes are supported by wglCreatePbufferARB:

...

WGL_TEXTURE_TARGET_ARB

This attribute indicates the target for the texture that will be created when the pbuffer is created with a texture format other than WGL_NO_TEXTURE_ARB. This attribute can be set to WGL_NO_TEXTURE_ARB, WGL_TEXTURE_1D_ARB, WGL_TEXTURE_2D_ARB, WGL_TEXTURE_CUBE_MAP_ARB, or WGL_TEXTURE_RECTANGLE_NV. The default value is WGL_NO_TEXTURE_ARB.

(Modify power-of-two error for wglCreatePbufferARB)

```
ERROR_INVALID_DATA     The pixel format attribute
                       WGL_TEXTURE_TARGET_ARB is WGL_TEXTURE_1D_ARB,
                       WGL_TEXTURE_2D_ARB, or
                       WGL_TEXTURE_CUBE_MAP_ARB, and WGL_PBUFFER_WIDTH
                       and/or WGL_PBUFFER_HEIGHT is not a power of
                       two.
```

(Add new wglCreatePbufferARB error)

```
ERROR_INVALID_DATA     WGL_TEXTURE_TARGET_ARB is
                       WGL_TEXTURE_RECTANGLE_NV
                       and WGL_MIPMAP_TEXTURE_ARB is non-zero.
```

(Add wglCreatePbufferARB errors missing from the ARB_render_texture spec)

```
ERROR_INVALID_DATA     WGL_TEXTURE_FORMAT_ARB is WGL_TEXTURE_RGB_NV,
                       WGL_TEXTURE_TARGET_ARB is
                       WGL_TEXTURE_RECTANGLE_NV, and the
                       WGL_BIND_TO_TEXTURE_RECTANGLE_RGB_NV attribute
                       is not set in the pixel format.

ERROR_INVALID_DATA     WGL_TEXTURE_FORMAT_ARB is WGL_TEXTURE_RGB_NV,
                       WGL_TEXTURE_TARGET_ARB is not
                       WGL_TEXTURE_RECTANGLE_NV, and the
                       WGL_BIND_TO_TEXTURE_RGB_NV attribute is not set
                       in the pixel format.

ERROR_INVALID_DATA     WGL_TEXTURE_FORMAT_ARB is WGL_TEXTURE_RGBA_NV,
                       WGL_TEXTURE_TARGET_ARB is
                       WGL_TEXTURE_RECTANGLE_NV, and the
                       WGL_BIND_TO_TEXTURE_RECTANGLE_RGBA_NV attribute
                       is not set in the pixel format.

ERROR_INVALID_DATA     WGL_TEXTURE_FORMAT_ARB is WGL_TEXTURE_RGBA_NV,
                       WGL_TEXTURE_TARGET_ARB is not
                       WGL_TEXTURE_RECTANGLE_NV, and the
                       WGL_BIND_TO_TEXTURE_RGBA_NV attribute is not
                       set in the pixel format.
```

Modify wglBindTexImageARB (only adding verbiage for supporting texture rectangles):

The command

```
BOOL wglBindTexImageARB (HPBUFFERARB hPbuffer, int iBuffer)
```

defines a one-dimensional texture image, a two-dimensional texture image, a two-dimensional texture rectangle image, or a set of two-dimensional cube map texture images...

The texture targets are derived... If the texture target is WGL_TEXTURE_2D_ARB, then <iBuffer> defines a 2D texture for the current 2D texture object. If the texture target is WGL_TEXTURE_RECTANGLE_NV, then <iBuffer> defines a texture rectangle for the current texture rectangle object. If the texture target is WGL_TEXTURE_1D_ARB, then <iBuffer> defines a 1D texture for the current 1D texture object.

New State

None

Name

    NV_swap_group

Name Strings

    WGL_NV_swap_group

Status

    Shipping since 2003 on Quadro GPUs with framelock support

Version

    Date: 02/20/2008   Revision: 1.0

Number

    351

Dependencies

    Written based on the wording of the GLX_SGIX_swap_group and
     GLX_SGIX_swap_barrier specifications.

    WGL_EXT_swap_control affects the definition of this extension.
    WGL_EXT_swap_frame_lock affects the definition of this extension.

Overview

    This extension provides the capability to synchronize the buffer swaps
    of a group of OpenGL windows. A swap group is created, and windows are
    added as members to the swap group. Buffer swaps to members of the swap
    group will then take place concurrently.

    This extension also provides the capability to sychronize the buffer
    swaps of different swap groups, which may reside on distributed systems
    on a network. For this purpose swap groups can be bound to a swap barrier.

    This extension extends the set of conditions that must be met before
    a buffer swap can take place.

Issues

    An implementation can not guarantee that the initialization of the swap
    groups or barriers will succeed because the state of the window system may
    restrict the usage of these features. Once a swap group or barrier has
    been sucessfully initialized, the implementation can only guarantee to
    sustain swap group functionality as long as the state of the window system
    does not restrict this. An example for a state that does typically not
    restrict swap group usage is the use of one fullscreen sized window per
    windows desktop.

New Procedures and Functions

```
    BOOL wglJoinSwapGroupNV(HDC hDC,
                            GLuint group);
    BOOL wglBindSwapBarrierNV(GLuint group,
                              GLuint barrier);

    BOOL wglQuerySwapGroupNV(HDC hDC,
                             GLuint *group);
                             GLuint *barrier);

    BOOL wglQueryMaxSwapGroupsNV(HDC hDC,
                                 GLuint *maxGroups,
                                 GLuint *maxBarriers);

    BOOL wglQueryFrameCountNV(HDC hDC,
                              GLuint *count);

    BOOL wglResetFrameCountNV(HDC hDC);
```

New Tokens

    none

Additions to the WGL Specification

Add to section 3.2.6, Double Buffering:

wglJoinSwapGroupNV adds <hDC> to the swap group specified by <group>.
If <hDC> is already a member of a different group, it is
implicitly removed from that group first. A swap group is specified as
an integer value between 0 and the value returned in <maxGroups> by
wglQueryMaxSwapGroupsNV. If <group> is zero, the hDC is unbound from its
current group, if any. If <group> is larger than <maxGroups>,
wglJoinSwapGroupNV fails.

wglJoinSwapGroupNV returns True if <hDC> has been successfully bound to
<group> and False if it fails.

wglBindSwapBarrierNV binds the swap group specified by <group> to <barrier>.
<barrier> is an integer value between 0 and the value returned in
<maxBarriers> by wglQueryMaxSwapGroupsNV. If <barrier> is zero, the group is
unbound from its current barrier, if any. If <barrier> is larger than
<maxBarriers>, wglBindSwapBarrierNV fails.
Subsequent buffer swaps for that group will be subject to this binding,
until the group is unbound from <barrier>.

wglBindSwapBarrierNV returns True if <group> has been successfully bound to
<barrier> and False if it fails.

wglQuerySwapGroupNV returns in <group> and <barrier> the group and barrier
currently bound to hDC, if any.

wglQuerySwapGroupNV returns True if <group> and <barrier> could be successfully
queried for <hDC> and False if it fails.
If it fails, the values of <group> and <barrier> are undefined.

wglQueryMaxSwapGroupsNV returns in <maxGroups> and <maxBarriers> the maximum
number of swap groups and barriers supported by an implementation which
drives window <hDC>.

wglQueryMaxSwapGroupsNV returns True if <maxGroups> and <maxBarriers> could be
successfully queried for <hDC> and False if it fails.
If it fails, the values of <maxGroups> and <maxBarriers> are undefined.

Before a buffer swap can take place, a set of conditions must be satisfied. The conditions are defined in terms of the notions of when a window is ready to swap and when a group is ready to swap.

Any hDC that is not a window (i.e. a non-visible rendering buffer) is always ready.

A window is ready when all of the following are true:

1. A buffer swap command has been issued for it.

2. Its swap interval has elapsed.

A group is ready when the following is true:

1. All windows in the group are ready.

All of the following must be satisfied before a buffer swap for a window can take place:

1. The window is ready.

2. If the window belongs to a group, the group is ready.

3. If the window belongs to a group and that group is bound to a barrier, all groups using that barrier are ready.

Buffer swaps for all windows in a swap group will take place concurrently after the conditions are satisfied for every window in the group.

Buffer swaps for all groups using a barrier will take place concurrently after the conditions are satisfied for every window of every group using the barrier, if and only if the vertical retraces of the screens of all the groups are synchronized. If they are not synchronized, there is no guarantee of concurrency between groups.

An implementation may support a limited number of swap groups and barriers, and may have restrictions on where the users of a barrier can reside. For example, an implementation may allow the users to reside on different display devices or even hosts.
An implementation may return zero for any of <maxGroups> and <maxBarriers> returned by wglQueryMaxSwapGroupsNV if swap groups or barriers are not available in that implementation or on that host.

The implementation provides a universal counter, the so called frame counter, among all systems that are locked together by swap groups/barriers. It is based on the internal synchronization signal which triggers the buffer swap.

wglQueryFrameCountNV returns in <count> the current frame counter for <swapGroup>.

wglQueryFrameCountNV returns TRUE if the frame counter could be successfully retrieved. Otherwise it returns FALSE.

wglResetFrameCountNV resets the frame counter of <swapGroup> to zero.

wglResetFrameCountNV returns TRUE if the frame counter could be successfully reset, otherwise it returns FALSE. In a system that has an NVIDIA framelock add-on adapter installed and enabled, wglResetFrameCountNV will only succeed when the framelock is configured as a Master system.

Errors

```
    wglJoinSwapGroupNV, wglQuerySwapGroupNV and wglQueryMaxSwapGroupsNV generate
    ERROR_DC_NOT_FOUND if <hDC> is not a valid HDC.
```

New State

    None

New Implementation Dependent State

    None

Name

    NV_video_output

Name Strings

    WGL_NV_video_output

Status

    Shipping since 2004 for NVIDIA Quadro SDI (Serial Digital Interface)

Version

    Last Modified Date: February 20, 2008

Number

    349

Dependencies

    OpenGL 1.1 is required.
    WGL_ARB_extension_string is required.
    WGL_ARB_pixel_format is required.
    WGL_ARB_pbuffer is required.

Overview

    This extension permits a color and or depth buffer of a pbuffer to
    be used for rendering and subsequent video output. After a pbuffer
    has been bound to a video device, subsequent color and or depth
    rendering into that buffer is displayed on the video output.

Issues

     1. Should the new pbuffer attributes be available through GL queries?

        No, like other pbuffer attributes you need to query them through the
        window system extension. This extension does not make any changes to
        OpenGL.

Implementation Notes

     1. Any created pbuffers must be the same resolution as that specified
        by the state of the video output device.

     2. Applications may use a single pbuffer or a collection of pbuffers
        to send frames/fields to a video device. In the first case, an
        application should block on the call to wglSendPbufferToVideoNV()
        to ensure synchronization. In the second caes, an application
        should utilize wglGetVideoInfoNV() in order to query vblank and
        buffer counters for synchronization.

Intended Usage

1) Configure the video output device via the NVCPL API or via the control panel which uses the NVCPL API.

2) Call wglChoosePixelFormatARB and find a suitable pixel format for rendering images. WGL_DRAW_TO_PBUFFER and one of WGL_BIND_TO_VIDEO_RGB_NV, WGL_BIND_TO_VIDEO_RGBA_NV or WGL_BIND_TO_VIDEO_RGB_AND_DEPTH_NV must be TRUE. The per-component pixel depth of the pbuffer must be equal to or greater than the per-component depth of the video output.

3) Create pbuffers and associated rendering contexts for each channel of video by calling wglCreatePbufferARB with one of WGL_BIND_TO_VIDEO_RGB_NV, WGL_BIND_TO_VIDEO_RGBA_NV or WGL_BIND_TO_VIDEO_RGB_AND_DEPTH_NV tokens in the attribute list set to TRUE. Set the width and height for each pbuffer to match that of the intended video output device.

4) Call wglGetVideoDeviceNV to retrieve the handles for all video devices available. A video device handle is required for each video stream.

5) Call wglBindVideoImageNV to bind each pbuffer drawable to a corresponding video device handle.

6) Start transfers on each video device using the appropriate NVCPL API function call.

7) Render the current frame/field for each stream to a pbuffer. Once rendering is complete, call wglSendPbufferToVideoNV() to send each frame/field to the video device.

9) Render subsequent video frames or fields calling wglSendPbufferToVideoNV() at the completion of rendering for each frame/field.

10) Stop transfers on the video device via the appropriate NVCPL API function call.

11) Call wglReleaseVideoImageNV to unbind each pbuffer drawable from its associated video device.

New Procedures and Functions

```
    DECLARE_HANDLE(HPVIDEODEV);

    BOOL wglGetVideoDeviceNV(HDC hDC, int numDevices,
                             HPVIDEODEV *hVideoDevice);

    BOOL wglReleaseVideoDeviceNV(HPVIDEODEV hVideoDevice);

    BOOL wglBindVideoImageNV (HPVIDEODEV hVideoDevice,
                              HPBUFFERARB hPbuffer, int iVideoBuffer);

    BOOL wglReleaseVideoImageNV (HPBUFFERARB hPbuffer, int iVideoBuffer);
    BOOL wglSendPbufferToVideoNV (HPBUFFERARB hPbuffer, int iBufferType,
                                  unsigned long *pulCounterPbuffer,
                                  BOOL bBlock);

    BOOL wglGetVideoInfoNV (HPVIDEODEV hpVideoDevice,
                            unsigned long *pulCounterOutputPbuffer,
                            unsigned long *pulCounterOutputVideo);
```

New Tokens

```
    Accepted by the <piAttributes> parameter of wglGetPixelFormatAttribivARB,
    wglGetPixelFormatAttribfvARB, and the <piAttribIList> and <pfAttribIList>
    parameters of wglChoosePixelFormatARB and wglCreatePbufferARB:

        WGL_BIND_TO_VIDEO_RGB_NV                        0x20C0
        WGL_BIND_TO_VIDEO_RGBA_NV                       0x20C1
        WGL_BIND_TO_VIDEO_RGB_AND_DEPTH_NV              0x20C2

    Accepted by the <iVideoBuffer> parameter of wglBindVideoImageNV and
    wglReleaseVideoImageNV:

        WGL_VIDEO_OUT_COLOR_NV                          0x20C3
        WGL_VIDEO_OUT_ALPHA_NV                          0x20C4
        WGL_VIDEO_OUT_DEPTH_NV                          0x20C5
        WGL_VIDEO_OUT_COLOR_AND_ALPHA_NV                0x20C6
        WGL_VIDEO_OUT_COLOR_AND_DEPTH_NV                0x20C7

    Accepted by the <iBufferType> parameter of wglSendPbufferToVideoNV:

        WGL_VIDEO_OUT_FRAME                             0x20C8
        WGL_VIDEO_OUT_FIELD_1                           0x20C9
        WGL_VIDEO_OUT_FIELD_2                           0x20CA
        WGL_VIDEO_OUT_STACKED_FIELDS_1_2                0x20CB
        WGL_VIDEO_OUT_STACKED_FIELDS_2_1                0x20CC
```

Additions to Chapter 2 of the OpenGL 1.2.1 Specification (OpenGL Operation)

None.

Additions to Chapter 3 of the OpenGL 1.2.1 Specification (Rasterization)

None.

Additions to Chapter 4 of the OpenGL 1.2.1 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 1.2.1 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 1.2.1 Specification (State and State Requests)

None.

Additions to the WGL Specification

```
    Add to the description of <piAttributes> in wglGetPixelFormatAttribivARB
    and <pfAttributes> in wglGetPixelFormatfv:

        WGL_BIND_TO_VIDEO_RGB_NV
        WGL_BIND_TO_VIDEO_RGBA_NV
        WGL_BIND_TO_VIDEO_RGB_AND_DEPTH_NV

    Add new table entries to match criteria in description of
    wglChoosePixelFormatARB:

        Attribute                           Type        Match Criteria

        WGL_BIND_TO_VIDEO_RGB_NV            boolean     exact
        WGL_BIND_TO_VIDEO_RGBA_NV           boolean     exact
        WGL_BIND_TO_VIDEO_RGB_AND_DEPTH_NV  boolean     exact

    Add wglGetVideoDeviceNV:

    The command

        BOOL wglGetVideoDeviceNV(HDC hDC, int numDevices,
                                 HPVIDEODEV hpVideoDevice)

     returns an array of upto <numDevices> handles to the available video
     devices in the array <hpVideoDevice>.  <numDevices> must be
     non-negative, and <hpVideoDevice> must not be NULL.

     It is not an error if the number of available video devices is larger
     that <numDevices>; in that case the first <numDevices> device handles
     are returned.  It is an error if <numDevices> is larger than the
     number of available video devices.  The order of devices returned in
     <hpVideoDevice> is implementation dependent.

     if wglGetVideoDeviceNV fails, FALSE is returned.  To get extended
     error information, call GetLastError.  Possible errors are as follows:

     ERROR_INVALID_HANDLE           <hDC> is not a valid handle.

     ERROR_INVALID_HANDLE           <hpVideoDevice> is NULL.

     ERROR_INVALID_VALUE            <numDevices> is negative.

     ERROR_INVALID_OPERATION        The video devices are not configured.

     ERROR_RESOURCE_NOT_AVAILABLE   The number of video devices requested
                                    are not available.


    Add wglReleaseVideoDeviceNV:

    The command

        BOOL wglReleaseVideoDeviceNV(HPVIDEODEV hVideoDevice)
    releases all resources associated with <hpVideoDevice>.

    If wglReleaseVideoDeviceNV fails, FALSE is returned.  To get extended
    error information, call GetLastError.  Possible errors are as follows:

    ERROR_INVALID_HANDLE            <hpVideoDevice> is not a valid handle.

    ERROR_INVALID_OPERATION         The video device is not allocated.


    Add wglBindVideoImageNV and wglReleaseVideoImageNV:

    The command

        BOOL wglBindVideoImageNV (HPVIDEODEV hpVideoDevice,
                                  HPBUFFERARB hPbuffer,
                                  int iVideoBuffer);

    binds <hPbuffer> to <hpVideoDevice> for subsequent scanout where
    <iVideoBuffer> specifies that <pbuffer> contains color, alpha or
    depth data.  Neither <pbuffer> nor <hpVideoDevice) can be NULL.

    If  wglBindVideoImageNV fails, FALSE is returned.  To get extended
    error information, call GetLastError. Possible errors are as follows:

    ERROR_INVALID_HANDLE            <hPbuffer> is not a valid handle.

    ERROR_INVALID_DATA              The pbuffer size is not correct.

    ERROR_INVALID_OPERATION         The video device is not yet configured.


    The command

       BOOL wglReleaseVideoImageNV (HPBUFFERARB hPbuffer,
                                    int iVideoBuffer);

    releases <hPbuffer> from a previously bound video device.  The
    parameter <iVideoBuffer> specifies that the pbuffer contains
    color, alpha or depth data.  <hPbuffer> may not be NULL.

    If  wglReleaseVideoImageNV fails, FALSE is returned. To get extended
    error information, call GetLastError. Possible errors are as follows:

    ERROR_INVALID_HANDLE            <hPbuffer> is not a valid handle.

    ERROR_INVALID_DATA              <iBuffer> is not a valid value.


    Add wglSendPbufferToVideoNV and wglGetVideoInfoNV:

    The command

       BOOL wglSendPbufferToVideoNV (HPBUFFER hPbuffer, int iBufferType,
                                     unsigned long *pulCounterPbuffer,
                                     BOOL bBlock);
```

    indicates that rendering to the <hPbuffer> is complete and that the
    completed frame/field contained within <hPbuffer> is ready for scan out
    by the video device where <iBufferType> specifies that <hPbuffer>
    contains the first field, second field, two stacked fields or a complete
    frame. <hPbuffer> cannot be NULL.
    An <iBufferType> of WGL_VIDEO_OUT_STACKED_FIELDS_1_2 indicates that
    <hPbuffer> does contain field1 and field2 with field1 in the upper half
    of <hPbuffer> and filed2 in the lower half, while
    WGL_VIDEO_OUT_STACKED_FIELDS_2_1 indicates field2 in the upper half
    of <hPbuffer> and filed1 in the lower half.
    The flag <bBlock> specifies whether or not the call should block until
    scan out of the specified frame/field is complete.
    <pulCounterPbuffer> returns the total number of frames/fields sent to
    the video device.

    If wglSendPbufferToVideoNV fails, FALSE is returned. To get extended
    error information, call GetLastError. Possible errors are as follows:

```
    ERROR_INVALID_HANDLE            <HPBUFFER> is not a valid handle.

    ERROR_INVALID_DATA              <iBufferType> is not a valid value.
```

    The command

```
        BOOL wglGetVideoInfoNV (HPVIDEODEV hpVideoDevice,
                                unsigned long *pulCounterOutputPbuffer,
                                unsigned long *pulCounterOutputVideo);
```

    returns in <pulCounterOutputVideo> the absolute count of vertical
    blanks on <hpVideoDevice> since transfers were started while
    <pulCounterOutputPbuffer> returns the count of the current pbuffer
    being scanned out by <hpVideoDevice>.

    If wglGetVideoInfoNV fails, FALSE is returned. To get extended error
    information, call GetLastError. Possible errors include:

```
    ERROR_INVALID_HANDLE             <hPVIDEODEVICE> is not a valid handle.
```

New State

    None

Usage Examples

    TBD

Revision History:

    20 February 2008
        public release