Topics

Rendering Pipeline

Shader Types

OpenGL Shader Language Basics

References

OpenGL Shading Language

John Kessenich, "The OpenGL Shading Language," OpenGL Language Version 4.20, Document Revision 6, August 2011.

OpenGL Commands for Shader Language Control

Mark Segal, Kurt Akeley, "The OpenGL Graphics System: A Specification (Version 4.3)", OpenGL, August 2012.

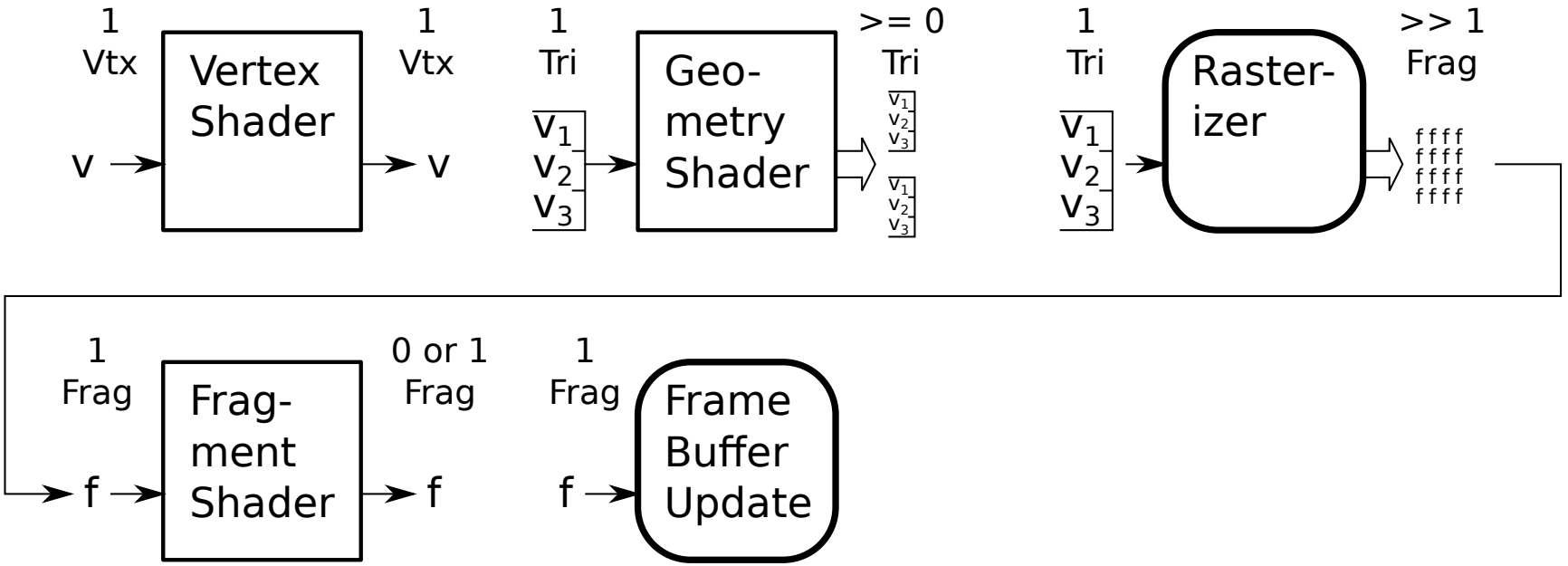
Pipeline:

An organization for software and hardware in which a fixed sequence of operations is carried out on data items...

Rendering Pipeline:

An organization for the set of steps needed to convert a set of vertices into a frame buffer image.

Simplified OpenGL Rendering Pipeline



OpenGL Shaders

Shader:

A program that performs certain rendering pipeline steps.

Major Rendering Pipeline Steps

Preparation

These activities are performed before invoking pipeline.

CPU specifies transforms, material properties, etc.

Calling, say, `glTranslatef`, helps set up pipeline...
... but does not start it running or feed it data.

Feed Data to Pipeline

Data enters in a unit including a vertex and its attributes.

This initiates the steps.

Vertex Processing Steps (By GPU for each vertex.)

- *Apply modelview transform to vertex.*

  Main result is vertex coordinate in eye space.

- *Compute lighted color of vertex.*

  Main result is lighted color.

- *Apply projection transform to eye-space vertex.*

  Result is vertex coordinate in clip space.

Primitive Assembly Steps

These steps operate on a primitive (a group of primitives).

- *Primitive Assembly (Group vertices into a primitive).*

  Result is, say, a group of 3 describing a triangle.

- *Clip (remove) off-screen parts of primitive.*

  Result is fewer and maybe different primitives.

- *Rasterize*

  Result is the set of fragments (fb locations) covered by primitive.

Fragment Processing Steps

These steps operate on a fragment.

- *Fetch texels, filter and blend.*

  Result is a frame-buffer ready color.

- *Frame Buffer Update*

  If fragment passes depth and other tests, write or blend.

Programmable Units

Programmable Unit:

Part of the pipeline that can be programmed (as defined by some API).

Choice of what is and isn't programmable constrained by:

Need to allow for parallel (multithreaded, SIMD, MIMD) execution.

Simple memory access.

Major OpenGL Programmable Units

Vertex Processor:

Transform vertex and texture coordinates, compute lighting.

Geometry Processor:

Using a transformed primitive and its neighbors generates new primitives. For example, replace one triangle with many triangles to more closely match a curved surface.

Fragment Processor:

Using interpolated coordinates, read filtered texels and combine with colors.

Languages

Shader:

A programmable part of a GPU. Name is now misleading but is still in common use.

Shader Language:

An language for programming shaders.

High-Level Shader Languages

OpenGL Shader Language

OpenGL standard.

Syntax very similar to C.

Language designed for vertex and fragment shaders.

Cg

Originated with ATI, adopted in Direct3D.

Syntax very similar to C.

Language designed for stream programs ...
... geometry, vertex, and fragment programs can be in stream form.

OpenGL Shader Language Important Features

C-like

CPP-like preprocessor directives.

Library of useful geometry functions.

Includes vector and matrix types and operators.

OGSL Data Types Example

Example

```
vec4 vertex_e = gl_ModelViewMatrix * o_point;
vec3 norm_e = gl_NormalMatrix * gl_Normal;
vec4 light_pos = gl_LightSource[1].position;
float phase_light = dot(norm_e, normalize(light_pos - vertex_e).xyz);
float phase_user = dot(norm_e, -vertex_e.xyz);
float phase = sign(phase_light) == sign(phase_user) ? abs(phase_light) : 0.0;
```

OGSL Storage Qualifiers

Storage Qualifiers

Used in a variable declaration, specifies where data stored.

Below, `in`, `uniform`, `constant`, and `out` are storage qualifiers.

```
in vec4 force;          // Input to this shader, different for each primitive.
uniform float x;        // Input to shader, value rarely changed.
const int sides = 5;    // Can never be changed.
out vec2 nudge;         // Output of this shader (input to some other).
```

Storage Qualifier Types

uniform:

Read-only by shader. Written by client, change is time consuming.

Typical use: transformation matrices.

in:

Input to shader. Read-only by shader that made the `in` declaration. Value is set either by client (using `glVertexAttrib` and friends) or by a prior stage shader (by writing an `out` variable.

Typical uses: vertex material properties (color), normal.

out:

Output of shader. Value is written by shader in which `out` declaration appears and read by shader in subsequent stage.

sampler:

Read-only by vertex and fragment shader. A handle to a texture unit, used by texture access functions.

Interpolation Qualifiers

Used for fragment shader inputs.

Specify how value should be interpolated.

flat:
No interpolation.

smooth:
Perspective-correct interpolation.

noperspective:
Linear interpolation.

Deprecated Storage Qualifiers.

These were used in earlier versions of OGSL.

They have been replaced by `in` and `out`.

attribute:
Deprecated. Like an `in` but only can be used for vertex shader.

varying:
Deprecated. When used in a vertex shader is the same as `out`, when used in a fragment shader is the same as `in`.

Storage Qualifier Example

```
// For vertex and fragment shaders:

uniform vec3 gravity_force;
uniform float gs_constant;
uniform vec2 ball_size;

// Vertex Shader Only

in float step_last_time;
in vec4 position_left, position_right, position_above, position_below;
in vec3 ball_speed;

out vec4 out_position;
out vec3 out_velocity;

// Fragment Shader Only
//
in vec4 out_position;
in vec3 out_velocity;
```

OGSL Functions

Function Parameters

OpenGL Shading Language 1.30 Section 4.4

Call by value.

Parameter Qualifiers:

in (default)

out

inout

Built In Functions

See OpenGL Shading Language 1.30 Section 8

Steps for adding a typical shader to existing OpenGL code:

Define what the shader is supposed to do.

Identify appropriate programmable units (vertex, geometry, fragment, etc).

Identify data that shaders will use.

If data from client (CPU) determine whether attribute or uniform.

For attributes and uniforms, determine if pre-defined or user-defined.

Write shader code.

In CPU code follow steps for installing shader. (*E.g.*, use `pShader`).

Get names of any new uniforms and attributes.

As necessary, initialize uniforms and attributes.

Turn shader on and off as necessary.

Example—Phong Shader

Phong Shader:

A lighting model in which the lighted color is computed at each fragment. (Otherwise the lighted color is computed at each vertex of a primitive and those lighted colors are interpolated across the fragments.)

Phong Shader Steps

- *Define what shader does.*
  Computes lighting at fragment using interpolated normal...

- *Identify appropriate units.*
  For computing lighting: fragment shader.

  For passing along normal and color info, vertex shader.

- *Identify data that shaders use.*
  VS: Lighting data, normal. (All pre-defined.)

  FS: Normal (interpolated), eye-space vertex coordinates. User def.

OpenGL Calls, from Initialization to Use (See OGL 4.3 Chapter 7)

Create Program Object (Once)

```
pobject = glCreateProgram()
```

For Each Shader (Vertex, Geometry, Fragment, etc.):

Create Shader Object

```
sobject = glCreateShader(GL_VERTEX_SHADER)
```

Provide Source Code to Shader Object and Compile

```
glShaderSource(sobject,1,&shader_text_lines,NULL);
glCompileShader(sobject);
```

Attach

```
glAttachShader(pobject,sobject);
```

Link (Once)

```
glLinkProgram(pobject);
```

Use (Many Times, *e.g.*, once per frame.)

```
glUseProgram(pobject);
```

Obtaining and Using Variable References

At run time variables identified by number.

At Initialization get **location** (index) of attributes and uniforms:

```
vsal_pinnacle = glGetAttribLocation(pobject,name);
sun_ball_size = glGetUniformLocation(pobject,name);
```

During Render (Infrequently) Change Uniform Value (Using location)

```
glUniform2f(sun_ball_size,ball_size,ball_size_sq);
```

During Render (Per Vertex Okay) Change Attribute Value (Using location)

```
glVertexAttrib4f(vsal_pinnacle,pinnacle.x,pinnacle.y,pinnacle.z,radius);
```

Done before each glVertex.

Same options as vertex, such as client and buffer object arrays.