Programmable GPUs Outline

Outline

EE 7700-1 Lecture Transparency. Formatted 11:30, 25 March 2009 from set-prog-api.

References

papi-2

OpenGL

OpenGL Shading Language

John Kessenich, “The OpenGL Shading Language,” OpenGL Language Version 1.20, Document Revision 8, September 2006.

OpenGL Commands for Shader Language Control

Mark Segal, Kurt Akeley, “The OpenGL Graphics System: A Specification (Version 2.1)”, OpenGL, July 2006.

papi-2

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.

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. (Not in OpenGL 2.1, defined in extensions.)

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.

Shader Assembly Language:
An assembly-like language for programming GPUs.

High-Level Shader Language:
A high-level language for programming GPUs.

Shader Assembly Language

At one time, only way to program.

Unlike a true assembly language . . .
. . . no instruction encoding defined . . .
. . . no promise of a one-to-one correspondence with machine instructions.

Translated into machine instructions (or micro-instructions) by API implementation.

Many APIs not picky about matching assembly language to target.

Currently might be used for tuning code from high-level shader language.

Separate languages defined for vertex, geometry, and fragment processors.

Early languages closely match underlying hardware, so more useful for performance tuning.

Defined as OpenGL extensions.

First-Generation Languages

NV_vertex_program (For vertex processor)

Close match to GEForce 3 hardware.

No branches or memory (texture or otherwise) access.

NV_fragment_program (2003) (For fragment processor)

Arbitrary texel access. (Can ignore or modify provided texture coords.)

Instructions for texture access and interpolation.

No branching.

Shader Assembly Languages

papi-7

Second-Generation Languages

NV_vertex_program2, NV_vertex_program3.

NV_fragment_program2,

Later Languages

Full support for integer operations, branching.

NV_gpu_program4: Instructions common to each kind of shader.

NV_vertex_program4.

NV_geometry_program4.

NV_fragment_program4.

High-Level Shader Languages

OpenGL Shader Language

OpenGL standard.

Syntax very similar to C.

Language designed for vertex and fragment shaders.

Current version is 1.3.

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 (OGSL)

papi-9

OpenGL Shader Language Important Features

C-like

CPP-like preprocessor directives.

Library of useful functions.

OGSL Data Types

papi-10

Data Types

OpenGL Shading Language 1.30 Section 4.1

Scalar types: bool, int, float

Vectors of bool, int, float.

- Element access: xyzw, rgba, stpq. E.g., var.xy, var.r

Matrices of float.

Structures

OGSL Data Types

papi-11

Integer

Signed and unsigned.

Thirty-two bits.

Earlier versions had lower precision and lacked bitwise operations.)

papi-11

OGSL Data Types

papi-12

Float

IEEE 754 Single Format

Calculations may be to less than IEEE 754 precision.

papi-12

OGSL Data Types

papi-13

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;
```


papi-13

OGSL Data Types

papi-14

Variable Types

Uniforms:

Read-only by shader. Changed by client, change is time consuming. Implemented as shader constants.

Attributes:

Read-only by vertex shader, not available to fragment shader. Changed by client, change is fast.

Varying:

Written by vertex shader, interpolated for fragment shader where read-only.

Sampler:

Read-only by vertex and fragment shader. Value is a filtered texel.

papi-14

OGSL Data Types

papi-15

Storage Qualifiers

const

attribute

Read only.

Not allowed in fragment shaders.

uniform

Read only.

varying

Written by vertex shader.

Interpolated for fragment shader.

Read only for fragment shaders.

Storage Qualifier Example

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

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

varying vec4 out_position;
varying vec3 out_velocity;
```

OGSL Functions

papi-17

Function Parameters

OpenGL Shading Language 1.30 Section 4.4

Call by value.

Parameter Qualifiers:

- in (default)
- out
- inout

OGSL Functions

papi-18

Built In Variables

OpenGL Shading Language 1.30 Section 7

Pre-defined uniform, attribute, and varying variables.

papi-18

OGSL Functions

papi-19

Built In Functions

See OpenGL Shading Language 1.30 Section 8

OGSL Use

papi-20

Code Use Overview

Suppose something (tube) needs special lighting.

Shader language code in light.cc.

All steps below done by code using OpenGL.

- Initialize step: Load, compile, and link light.cc.
- During render, when ready for tube: Install light.cc.
- As needed, write uniform values.
- At this point all vertices handled by light.cc.
- When done with tube install another shader or switch to fixed func.

OGSL Use

papi-21

See OpenGL 2.1 Section 2.15

Initialize Program

Create Shader Object

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

Provide Source Code to Shader Object

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

Compile Shader Object

```
glCompileShader(sobject);
```

Attach and Link

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

Use

```
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.

OGSL Vertex Shader Examples

papi-23

Vertex Shader Examples

Minor variation on lighting.

Compute geometry of bump and circle.

Physics

OGSL Vertex Shader Example

papi-24

Example: Variation on Lighting

Program: cube4.cc (gpu acceleration off)

Shader Code: cube4_vshader.cc::vs_main_lighting()

Why: Tweak lighting.

Notes:

Shader computes transformation, lighting, and texture coordinates.

Program switches between vs_main_lighting and fixed func.

OGSL Vertex Shader Example

Example: Compute Geometry

Program: cube4.cc (gpu acceleration on)

Why: Less work for CPU.

Shader Code: cube4_vshader.cc::vs_main_circle() and vs_main_bump()

Notes:

Not a geometry shader.

Program switches between vs_main_circle, vs_main_bump and fixed func.

OGSL Vertex Shader Example

papi-26

Example: Physics

Program: cube5.cc

What: Shader time-steps lattice of masses.

Why: Less work for CPU.

papi-26

Sample Program cube4.cc

papi-27

Sample Program cube4.cc

Displays a rotating cube.

Cube faces have textures: syllabus, pic of EE building, etc.

Ball bouncing around cube.

Low-speed impact: color circle.

High-speed impact: bump.

Vertex Shader Uses

Lighting tweak.

Circle painting.

Bump painting.

EE 7700-1 Lecture Transparency. Formatted 11:30, 25 March 2009 from set-prog-api.

papi-27

Sample Program cube4.cc

papi-28

Data Representations

Cube:

Admittedly messy part of code.

Cube position: transformation matrix in `pCube`.

Textures: `pCube_Face_Info` (6-element array).

History of ball contact: `pContact_List` (6-element array).

Ball: position, speed, size.

papi-28

Sample Program cube4.cc

papi-29

Physics

At each time step . . .
. . . move cube to new position . . .
. . . move ball to new position.

To move cube: update rotation matrix using time and spin rate.

To move ball:

Find next intercept of ball trajectory with cube face.

If intercept after end of time step, done.

Record intercept position and ball velocity.

Recompute ball trajectory and repeat.

papi-29

Sample Program cube4.cc

papi-30

Graphics

Trivial Case: no translucency and ball doesn't leave marks:

Render cube faces and ball.

Middle Case: no translucency but ball does leave marks:

Stencil holes at collision points.

Render face using stencil to leave hole positions unchanged.

Render bumps.

Render ball.

Code as Written: translucency and ball leaves marks:

Sort faces so that face never under one already rendered.

Render bumps before face if bump behind face.

papi-30

Sample Program cube4.cc

papi-31

Code Organization

Initialization: Set idle callback.

Idle Callback: Wait for beginning of display refresh (or 30ms) . . .
. . . request redisplay.

Redisplay: Advance physics (time step), then render frame.

papi-31

Sample Program cube4.cc

papi-32

Use of Vertex Shaders

Lighting (`vs_lighting`). Used for cube faces (except marks).

Circle (`vs_circle`). Used for stenciling and drawing circles.

Bump (`vs_bump`). Used for drawing bump.

papi-32