Programming with OpenGL
Shaders 1

Adapted From:
Ed Angel
Professor of Emeritus of Computer Science
University of New Mexico
Objectives

Shader Programming Basics
Simple Shaders
  Vertex shader
  Fragment shaders
Shaders

What is a shader (and why is it called that?)

How might you use a Vertex Shader?

A Fragment Shader?
Vertex Shader Applications

Per Vertex Computation

Lighting Calculations
  - More realistic models
  - Cartoon shaders
Moving/Transforming vertices
  - Morphing
  - Procedural Deformation
  - Skinning

Fragment Shaders

*Per Fragment Computation*

- Per fragment lighting calculations
- Blending, compositing

per vertex lighting

per fragment lighting
Fragment Shaders

Texture mapping

smooth shading  environment mapping  bump mapping
Shaders History

First programmable shaders were programmed in an assembly-like manner (mid 90’s ??)

- OpenGL extensions added for vertex and fragment shaders
- Cg (C for graphics) C-like language for programming shaders
  Work(ed) with both OpenGL and DirectX
  (2012 deprecated)

OpenGL Shading Language (GLSL)

As of OpenGL 3.1, application must provide shaders (vertex and fragment)
Data Types

C types: float, double, int, uint, bool

Vectors: float, double, int, uint bool  [2, 3, 4]
   vec2, dvec2, ivec2, uvec2, bvec2

Matrices: float and double only [2x2 up to 4x4]
   mat2x2, dmat2x2
   Stored by columns (column major)
   Standard referencing m[row][column]

C++ style constructors
   vec3 a = vec3(1.0, 2.0, 3.0)
   vec2 b = vec2(a)
Execution Model: vertex portion

Vertex data

Shader Program

init()

Application Program

mainloop()

glDrawArrays

GPU Memory

Vertex Shader

GPU

Primitive Assembly

Vertex
Storage Qualifiers

cost: read-only as in C

in: input to a shader stage
    from a previous stage
    from the application as vertex attributes

out: outputs from a shader stage
    e.g. final fragment color from fragment shader

uniform: value will be specified by application before shader executes AND does not change across the primitive
    shared between all stages
    must be declared as globals
    e.g. color for an entire primitive such as a triangle
IN and OUT Qualified

IN: User defined (in application, earlier shader stage)

Use *in* qualifier to get to shader

- `in float temperature`
- `in vec3 velocity`

OUT: User defined (in shader)

Use *out* qualifier to get out of a shader

- `out vec3 color`
**Simple Vertex Shader (pass through)**

```glsl
in vec4 vPosition;
void main(void) {
    gl_Position = vPosition;
}
```

- **input from application**
- **must link to variable in application**
- **built in variable**
Built-In Variables

The **OpenGL Shading Language** defines a number of special variables for the various shader stages. These **predefined variables** (or built-in variables) have special properties. They are usually for communicating with certain fixed-functionality. By convention, all predefined variables start with "gl_"; no user-defined variables may start with this.

http://www.opengl.org/wiki/Built-in_Variable_%28GLSL%29
Execution Model: fragment portion

Application Program → Shader Program → GPU

GPU:
- Rasterizer
  - Fragment Shader
    - Frame Buffer

Fragment Shader:
- Fragment
  - Frame Buffer

Fragment Color
Simple Fragment Shader

out vec4 fragColor;

void main(void)
{
  fragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
Uniform Qualified

Variables that are constant for an entire primitive
Can be changed in application and sent to shaders
Cannot be changed in shader
Used to pass information to a shader
  e.g. primitive bounding box
  e.g. constant colors without interpolation
Uniform Qualified Example

Gluint faceColorParam;
faceColorParam = glGetUniformLocation(myProgObj, "faceColor");

/* faceColor defined in shader */

GLfloat faceColor[3];
/* faceColor set in application */
faceColor[0] = 1.0
faceColor[1] = faceColor[2] = 0.0;

/* link app var to shader var */
glUniform3fv(faceColorParam, faceColor);
/* 1D float*/
Interpolating Variables

Variables that are passed from vertex shader to fragment shader

- OUT variables in Vertex Shader
- IN variables in Fragment Shader

Automatically *interpolated* over the primitive by the rasterizer

http://www.geometrian.com/programming/tutorials/graphicspipeline/opengl_4_rasterization.php
Passing values in functions

Can have multiple functions in a shader

Call by \textit{value-return}

\textbf{IN}

for function arguments copied into a function

\textbf{OUT}

values copied out into the calling arguments
undefined upon entrance

\textbf{INOUT}

value copied into from arguments and out of the function back to the arguments
Operators and Functions

Standard C functions

   Trigonometric
   Arithmetic
   Normalize, reflect, length

Overloading of vector and matrix types

   mat4 a;
   vec4 b, c, d;
   c = b*a; // a column vector stored as a 1d array
   d = a*b; // a row vector stored as a 1d array
Swizzling and Selection

Can refer to array elements by element using [] or selection (. ) operator with

x, y, z, w [position elements]
r, g, b, a [color elements]
s, t, p, q [texture elements]

a[2], a.b, a.z, a.p are the same

**Swizzling** operator lets us manipulate components

```cpp
vec4 a;
a.yz = vec2(1.0, 2.0);
```

http://en.wikipedia.org/wiki/Pointer_swizzling
And more…

Conditional
Flow control
Loops
Structs
...etc.