Programming with OpenGL
Shaders I

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

Shader Basics
Simple Shaders
  Vertex shader
  Fragment shaders
Vertex Shader Applications

Computations for each vertex

Moving vertices
- Morphing
- Procedural Deformation
- Skinning

Lighting Calculations
- More realistic models
- Cartoon shaders
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
Writing Shaders

First programmable shaders were programmed in an assembly-like manner

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

OpenGL Shading Language (GLSL)
GLSL Basics

Part of OpenGL 2.0 and up
High level C-like language
New data types
   Matrices
   Vectors
   Samplers
As of OpenGL 3.1, application must provide shaders (vertex and fragment)
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**
Execution Model

Vertex data
Shader Program

Application Program

GPU

Vertex Shader

Primitive Assembly

glDrawArrays

Vertex

Simple Fragment Shader

out vec4 fragColor;

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

*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

1. **Application**
2. **Shader Program**
3. **Frame Buffer**
4. **Fragment Shader**
5. **Rasterizer**
6. **Fragment**
7. **Fragment Color**
**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
   Standard referencing m[row][column]

C++ style constructors
   vec3 a = vec3(1.0, 2.0, 3.0)
   vec2 b = vec2(a)
**Storage Qualifiers**

const: 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 Qualified

User defined (in application program)

Use **in** qualifier to get to shader

```glsl
in float temperature
in vec3 velocity
```
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

```c
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;

/* set value of the variable in shader*/
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* by the rasterizer

http://www.geometrian.com/programming/tutorials/graphicspipeline/opengl_4_rasterization.php
Example: Vertex Shader with hardcoded color

```glsl
in vec4 vPosition;
out vec4 color;

const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
void main(void)
{
    gl_Position = vPosition;
    color = red;
}
```
Example: Passthrough Fragment Shader

```glsl
out vec4 fragColor;
in vec4 color;

void main()
{
    fragColor = color;
}
```
Passing values

Can have multiple functions in a shader

Call by value-return

IN

for function parameters copied into a function

OUT

values copied out

undefined upon entrance

INOUT

value copied into and out of the function
Operators and Functions

Standard C functions

- Trigonometric
- Arithmetic
- Normalize, reflect, length

Overloading of vector and matrix types

```c
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

\[ x, y, z, w \]
\[ r, g, b, a \]
\[ s, t, p, q \]
\[ a[2], a.b, a.z, a.p \] 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.