Introduction to Shaders

Santa Claus

GLEW – A New Include

The version of OpenGL we are using is a bit old, so we need to use an extra library - GLEW (OpenGL Extension Wrangler Library).

The include for GLEW is

```
#include <GL/glew.h>
```

and **must** come before any other GL related includes.

Before using any GLEW commands you must call the function `glewInit()`.

The Makefile for build script will need to have `-lGLEW` added to it.

Make Space – A Shader is Coming

Like with any other OpenGL object, before we start doing anything with it, we first need to ask openGL to give us a shader. To do this, we need an int to represent the created shader, and a to know what sort of shader we want.

```
GLhandleARB vs =
    glCreateShaderObjectARB(GL_VERTEX_SHADER);
GLhandleARB fs =
    glCreateShaderObjectARB(GL_FRAGMENT_SHADER);

if(vs == 0 || fs == 0){
  printf("Could not create shaders\n");
  exit(EXIT_FAILURE);
}
```

Read The Code - Get the Source Code

Shaders are not compiled by g++. They are read into your program and compiled at run time. As such we need to first get the whole shader program as a `char*`, and then pass that in to GLEW to make a shader out of.

```
char* readFile(char*);
/* Code for readfile*/

char* vsCode = readFile("vertex.glsl");
char* fsCode = readFile("fragment.glsl");
```

Read The Code - Attach to Shader

We use glShaderSourceARB to attach the source code to the shader. The arguments are:

- The shader to attach the code to
- The number of `char*`s that make up the source code
- The array of `char*`s that are the source code
- Array of lengths of the `char*`s that are the source code, or NULL if they are NULL terminated

```
glShaderSourceARB(vs,1,&vsCode,NULL);
glShaderSourceARB(fs,1,&fsCode,NULL);
```

Compile The Shader – Make a Program

To compile our shaders we just run them through the shader compiler. This would seem to be the easiest step.

```
glCompileShaderARB(vs);
glCompileShaderARB(fs);
//checkForErrors(vs);
//checkForErrors(fs);
```

Attaching The Shaders – Programs

Shaders aren't just used on their own. Instead you attach them to a `Programs` and use those. Note that you have to link the shaders after attaching them!

```
GLhandleARB program = glCreateProgramObjectARB();
glAttachObjectARB(program, vs);
glAttachObjectARB(program, fs);

glLinkProgramARB(program);
//checkForErrors(program);
```

Enable The Shader — FIRE!

Using a shader is very easy. You simply use the given program. Note that using program 0 will disable it.

```
glUseProgramObjectARB(program);
//Shader is now active
/*
 Code for drawing stuff in here
/*
glUseProgramObjectARB(0);
//Shader is now inactive
```

Finding Errors

We want to check for any errors, and print out the relant information for a shader or program.

```
void checkForErrors(GLhandleARB obj){
  int infologLength=0, charsWritten=0;
  char *infoLog;
  glGetObjectParameterivARB(obj,
       GL_OBJECT_INFO_LOG_LENGTH_ARB, &infologLength);
  if (infologLength > 0){
        infoLog = (char*)malloc(infologLength);
        glGetInfoLogARB(obj, infologLength,
                &charsWritten, infoLog);
        printf("%s\n",infoLog);
        free(infoLog);
    }
}
```

GLSL — Talking Shader Language

GLSL, the language for writing shaders is based on C. There are two kinds of shaders we care about - Vertex and Fragment (there are also geometry and tessellation shaders).

Shaders, similar to C programs start with a main method.

```
void main(){
 //your code goes here...
}
```

Vertex Shaders — Hello World

Once we start making our own shaders, the default behaviours that you are used to from OpenGL are no longer there. So the first step is to learn how to replicate some of the original behaviour of OpenGL. Here we will write a basic vertex shader that does nothing more than the default transformatoins.

```
void main(){
 /*gl_position = gl_ProjectionMatrix *
       gl_ModelviewMatrix * gl_Vertex;
 gl_position = gl_ModelViewProjectionMatrix *
       gl_Vertex;*/
 gl_position = ftransform(); //do the normal thing
}
```

Fragment Shaders — Hello World

Here we will make a basic fragment shader. All it does is set the color of the current fragment.

```
void main(){
 gl_FragColor = vec4(1.0,0,0,1.0);
}
```

Types in GLSL

Before we do anything in GLSL we first need to know what types it has. Fortunately we have a table of ones we should think about.

	vec2	mat2
float	vec3	mat3
	vec4	mat4

Table 1: Some GLSL types

Some examples:

```
vec4 t = vec4(1.0, 2.0, 3.0, 4.0);
vec4 a = t.xyzw; //t.rgba or t.stpq
vec4 b = t.wzyx; // backwards
vec2 c = t.zz;
```

Vertex Properties

Name	Explanation	Type
gl_Vertex	Position	vec4
gl_Normal	Normal	vec4
gl_Color	Color of vertex	vec4
gl_MultiTexCoord0	Texture coordinate of texture unit 0	vec4
gl_MultiTexCoord1	Texture coordinate of texture unit 1	vec4
gl_MultiTexCoord2	Texture coordinate of texture unit 2	vec4
gl_MultiTexCoord3	Texture coordinate of texture unit 3	vec4
gl_MultiTexCoord4	Texture coordinate of texture unit 4	vec4
gl_MultiTexCoord5	Texture coordinate of texture unit 5	vec4
gl_MultiTexCoord6	Texture coordinate of texture unit 6	vec4
gl_MultiTexCoord7	Texture coordinate of texture unit 7	vec4
gl_FogCoord	Fog Coord	float

Table 2: OpenGL variables per vertex

Variable Visibility

There are two kinds of variable annotations that we are going to look at `uniform` and `varying` (we are ignoring attribute). Note that these are the types of global (in shader land) variables, rather than method or argument ones.

Uniform variables hold a constant value that is set in the C++ code and simply used in the shader.

Varying variables share values between vertex and fragment shaders. Note, that the value of the varying variable is interpolated by OpenGL when given to the fragment shader.

A Better Simple Shader

Lets now make shaders that interpolate colors.

```
varying vec4 currentColor;
void main(){
 currentColor = gl_Color;
 gl_position = ftransform();
}
```

Figure 1: Vertex Shader

```
varying vec4 currentColor;
void main(){
  gl_FragColor = currentColor;
}
```

Figure 2: Fragment Shader

Uniform Variables

Uniform values are constant and set in the C++ code. So you need a way to affect the shader from inside C++. This is done is two steps. First you get a handle for the variable you are interested in (by name), and then set its value.

```
GLint loc = glGetUniformLocation(program, "name");
if (loc != -1){
    glUniform1f(loc, 0.432);
}
```

Contents