Introduction

Because OpenGL 3.x and up is extremely shader oriented, I am going to show you how to integrate our shader code from the GLSL section of the website. I am not going to go into GLSL details here, as I have done that, and will continue to do that in the GLSL portion of the site. This tutorial is a quick guide on how to get shaders working in our current framework. So why don't we get to it, so we can move on to using matrices in the new OpenGL pipeline. Keep in mind that I am only working with Vertex and Fragment shaders still, I will introduce Geometry shaders in a later tutorial.

Coding

The code for this tutorial is going to be fairly minimal, and I may have updates to the shader.cpp and shader.h files that are in the GLSL portion of the site. If you have any queries, feel free to ask.

The first thing we are going to do, is copy our shader.h and shader.cpp files from the GLSL section of the site, and add them into our new Visual Studio project. Then go ahead and open opengl_3.h.

Opengl_3.h

Inside of opengl_3.h where we have our opengl context class structure, jump up the top and make sure you include the shader.h file into your source code.

```
#include "shader.h"
```

Once we have done this, go down into our private variables for our context class, and add an instance of a shader variable. If you are working with several shaders, you can make this into an array, or create separate shader variables. Personally I would create an array of shaders.

```
class OpenGLContext {
...
private:
	int windowWidth; // Store the width of our window
	int windowHeight; // Store the height of our window

	Shader *shader; // Our GLSL shader
...
};
```

Opengl_3.cpp

Once we have a shader variable declared that we can play with, open up opengl_3.cpp and jump down to the setupScene method. Inside of here we are going to set our shader to a new shader object, based on shader.vert and shader.frag. Both of which are included in the project.

```
void OpenGLContext::setupScene(void) {
	glClearColor(0.4f, 0.6f, 0.9f, 0.0f); // Set the clear color based on
Microsofts CornflowerBlue (default in XNA)

	shader = new Shader("shader.vert", "shader.frag"); // Create our shader by
loading our vertex and fragment shader
}
```

And then all we have left to do is jump to our renderScene method and bind and unbind our shader as required. You will notice that we don't have any geometry to draw yet, so you won't see anything yet, but don't worry, we are getting there :)

```
void OpenGLContext::renderScene(void) {
    glViewport(0, 0, windowWidth, windowHeight); // Set the viewport size to fill the
window
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT); //
Clear required buffers

       shader->bind(); // Bind our shader

       shader->unbind(); // Unbind our shader

       SwapBuffers(hdc); // Swap buffers so we can see our rendering
}
```

Shader.cpp

I mentioned at the start that these are pretty much the same shader.cpp and shader.h files that are in the GLSL section of the website. Here I am going to list one major change that I have made. I am binding two variables to our shader. These variables are constants and are called in_Position and in_Color and will be used to send positions and colors to our shader without binding and unbinding them every frame. These lines are added in our init method for our shader:

```
void Shader::init(const char *vsFile, const char *fsFile) {
...
       glBindAttribLocation(shader_id, 0, "in_Position"); // Bind a constant attribute
location for positions of vertices
       glBindAttribLocation(shader_id, 1, "in_Color"); // Bind another constant
attribute location, this time for color
...
}
```

Shader.vert

Because this tutorial has been so short, I am going to give a quick talk about the changes done to vertex and fragment shaders in GLSL 1.50. To declare which version of GLSL you are using, you can use a line similar to the following, which sets our GLSL version to 1.50:

#version 150 core

This can be changed for any valid version of GLSL. In GLSL 1.50, we no longer have gl_ModelViewMatrix, gl_ProjectionMatrix, gl_Vertex, etc, etc. Because we now have to manage and send all of this information to the shaders ourselves. In the next tutorial I will introduce self manipulated model, view and projection matrices, but for now I will stick with the basics. Because I created the two variables in_Position and in_Color to bind to our shader, we have to create these. They will both be vec3 variables, and because we are using GLSL 1.5, we also need to declare if the variable is going "in" or "out" of the current shader. There are no more uniform or varying variables to deal with. I am also going to create one "out" variable called pass_Color which is also a vec3 variable. This will send any outgoing color from the vertex shader to the fragment shader. Luckily setting the final position requires some similar code, which is setting the gl_Position variable.

#version 150 core

```
in vec3 in_Position;
in vec3 in_Color;

out vec3 pass_Color;

void main(void)
{
    gl_Position = vec4(in_Position, 1.0);
    pass_Color = in_Color;
}
```

Shader.frag

In our fragment shader, we are also going to declare the version of GLSL that we are working with and because we had an “out” variable in our shader.vert file, then we need an “in” variable of the same name in our fragment shader. So we are going to create an “in” variable, of type vec3 called pass_Color. Now fragment shaders no longer have gl_FragColor, instead we need to manually declare an “out” variable of type vec4, which will give our color to our GPU. I am going to call this “out” variable out_Color and I am going to set it to the pass_Color, which is the variable we sent in to our shader.

```
#version 150 core

in vec3 pass_Color;

out vec4 out_Color;

void main(void)
{
    out_Color = vec4(pass_Color, 1.0);
}
```

If all went well, you should now be set up to use shaders in your application and your application should run without any errors. Stick with me for the next two tutorials, and by the end of the second, we will now be rendering things to the screen. OpenGL 3.x and 4.x takes a bit more preparation, but it does make a difference in the end. If you have any questions you can comment below or email me at swiftless@gmail.com