CSCI 441 - Lab 09
Friday, October 28, 2016

LAB IS DUE BY **FRIDAY NOVEMBER 04 11:59 PM**!!

Today, we'll look at how to properly use shaders. We will look a simple example that does the minimum FFP requirements and then do a slight modification to the vertex shader (if desired).

Please answer the questions as you go inside your README.txt file.

Step 1 – A Generic Sphere

Type make and then run the lab. Hrmm, not quite working. If you double click the executable you'd get a message about a missing dll. The Makefile was supposed to copy this file for you but it doesn't (that's what the very last error is complaining about). You need to copy the C:/sw/opengl/bin/cygGLEW-1-13.dll file to the folder containing your executable. Now go ahead and run. You should see two whitish/greyish spheres. Our goal is to add a rainbow of color. One option would be to create the set of vertices ourselves and color each one. But that is very rigid and if we want to change the coloring, we then need to change the color of every vertex. Instead we will use a shader program to allow the colors to be generated dynamically.

There will be four files you need to modify for this lab:

- main.cpp
- include/Shader_Utils.h
- shaders/customShader.v.glsl
- shaders/customShader.f.glsl

Let's look at how a shader program gets properly compiled. Some of the existing code is there for you, other parts you will need to fill in. Go through it step by step. Open the file `include/Shader_Utils.h`. At the bottom of the file is the function `createShaderProgram()` that handles the compiling and linking of our program. Here are the important steps:

1. `compileShader()` – a function we write that takes a text file and a shader type. Sends the contents of the GLSL file to the GPU, staging it for compiling.

There are three things we need to do to compile a shader. First, we need to request a handle for our shader. Place this at TODO #1. *Hint: use the function glCreateShader().*

Reading the shader from file to a string is handled for you. The next step is to now send our code to the GPU. Do this at TODO #2. *Hint: use the function glShaderSource().*

Lastly, we need to have the GPU compile our shader. This will occur at TODO #3. *Hint: use the function glCompileShader().*

The shader log is then printed for you to check for errors.

2. Back inside `createShaderProgram()`, we are calling `compileShader()` twice. Once for our vertex shader and once for our fragment shader. Now it is time to actually create our shader program.

There are now again three steps to create our shader program. First, we need to request a handle for our program. Place this at TODO #4. *Hint: use the function glCreateProgram()*.

Next is to attach each shader to the program. Enter these TWO lines at TODO #5. *Hint: use the function glAttachShader()*.

Lastly, tell the GPU to link our shader program together at TODO #6. *Hint: use the function glLinkProgram()*.

The program log is then printed for you to check for errors.

Back in main.cpp, before we do any of our shader compilation we need to make sure we initialize GLEW. This is done through `glewInit()` (this occurs at LOOKHERE #1). Two things can go wrong when you try to compile the shaders if you had not done the proper set up.

First, if you do not include `<GL/glew.h>` then at compile time, you will get an error that `glShaderSource()`, et al. cannot be found. `glew.h` redefines `gl.h` internally, hence the reason it needs to be included before `gl.h` This redefinition includes all these shader specific functions. (This occurs at LOOKHERE #2)

Second, if you include `glew.h` but do not call `glewInit()` then you will get a Seg Fault at runtime when you try to call any of the shader functions. `glewInit()` sets up your OpenGL context to include the necessary extensions to work with shaders and the GPU.

In `main.cpp`, our `main()` function calls our user defined function `setupShaders()` and this is where we will call our Shader_Utils funciton. At TODO #7, call our createShaderProgram() function passing in the filenames of our two shader files: `shaders/customShader.v.glsl` and `.f.glsl`. Don't forget this returns our shader program handle so be sure to set our global variable.

We can now run our program. You may get an error from the GPU related to your shaders. You may not get an error and a sphere disappears. You may not get an error and you see the same two spheres as if nothing happened. Your result will vary (oh the joy of shader programming).

Let's now start with our GLSL code to create our shaders so we can actually see something interesting.

Step 2 – The Vertex Shader

Open `shaders/multitexturing.v.glsl`. Note one of our preprocessor directives

```
#version 120
```

This states what GLSL version the shader is written in. (*As an aside: The most recent version is 450 (for 4.50) but we are sticking with version 1.20 and OpenGL 2.1 since it is the most compatible version. And future versions are backward compatible. Once we get to GLSL 1.50 forces us to rewrite our entire OpenGL code and start using OpenGL 3.2. This rewrite involves passing every piece of information manually, including the ModelView matrix, Projection matrix, light information, etc. That doesn't provide us any real advantage at this stage. For the advanced graphics course, we will plan to focus on OpenGL 3.3+ and ensure our hardware is up to date. Continuing on.*)

The vertex shader has one main purpose that it must accomplish – we need to set gl_Position to our vertex in clip space. That means we need to do the transformation ourself. We have built in variables that we can use to do this calculation.

gl_ModelViewProjectionMatrix is a built-in uniform variable that contains our MVP matrix.

Q1: What is a uniform variable for GLSL?

gl_Vertex is a built-in attribute variable that contains our vertex location.

Q2: What is an attribute variable for GLSL?

Under Vertex Calculations at TODO #8, set gl_Position equal to the MVP matrix times the vertex. Order matters! The order in the sentence is the order you should set up your equation.

Great our vertex shader is done (though we'll come back to it). Onto the fragment shader!

Step 3 – The Fragment Shader

Open `shaders/multitexturing.f.glsl`. The Fragment Shader has one goal – to set the final RGBA value for this fragment. (*Aside, it can also modify the fragment's depth because we have not gotten to the depth test yet, but there's no reason we'll do that.*) For this Fragment Shader, the RGBA value will be constant. We need to set the built in variable, `gl_FragColor`. Let's set it equal to white, (1, 1, 1, 1) at TODO #10. (I know there's no #9…it got renumbered and I didn't want to shift everything down one)

And we're done! Let's head back to the comfort of C++ land to see if this compiles (it should).

Step 4 – Compiling the Shader

Rerun your program. If anything went wrong, you'll see output when the shader code gets compiled. If there is an error, modify the GLSL code and rerun the program.

Q3: Why do we not need to recompile the C++ code after we modify the GLSL code? Why are we allowed to rerun our C++ program to test new GLSL code?

When your shader code compiles successfully, the very last piece is to tell OpenGL to use our shader program instead of the Fixed Function Pipeline. Back in main.cpp at TODO #11, we need to use our shader program. Pass the handle to glUseProgram() right before we draw the first sphere. We want the second sphere to still be using the FFP, so turn shader programs off by passing zero (0) to the use program function before we draw the second sphere.

Compile and run. You should see one "sphere" that looks like a flat white circle and one sphere that has lighting being applied. Great, let's make our sphere a bit more colorful.

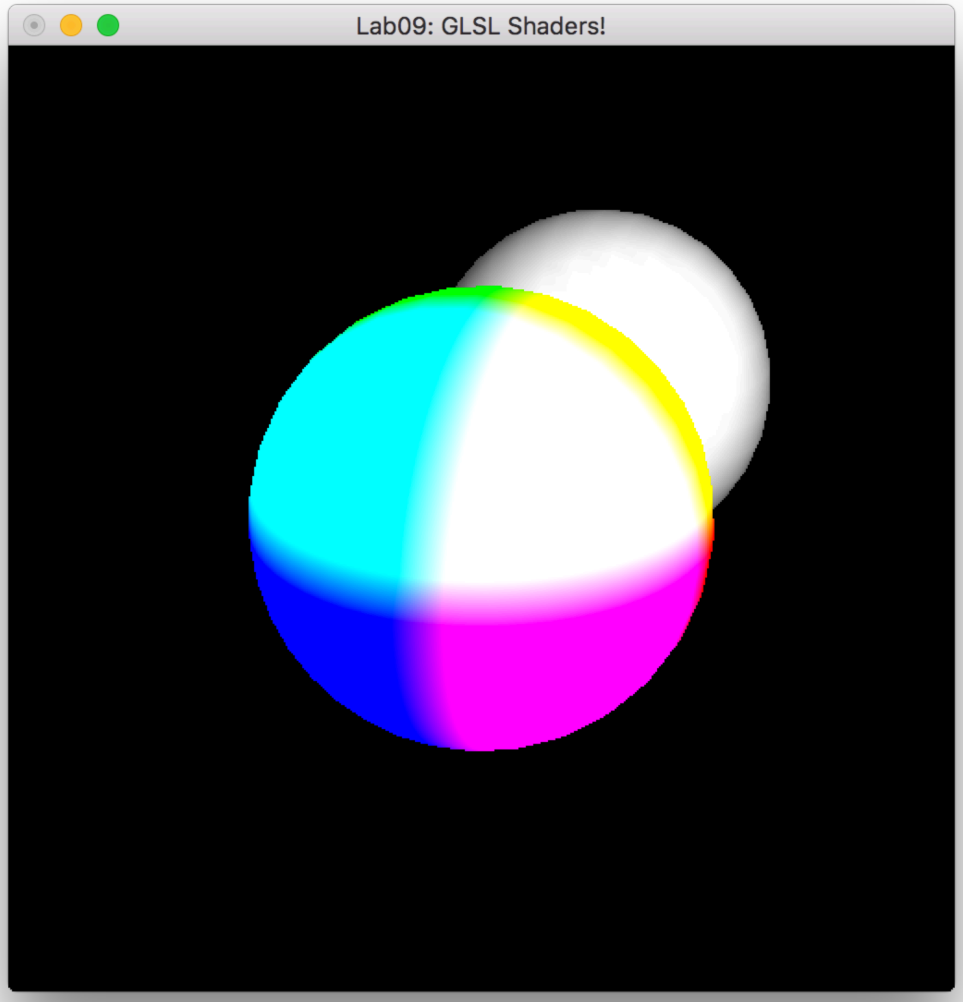
Step 5 – Pretty Our Shader Up

In our Fragment Shader, we don't want to set every fragment just to white. Instead, we'll have it based off the position of our vertex. Back to the Vertex Shader!

We need to pass information from our vertex shader to the fragment shader, so we will use a varying for this. Create a varying variable that will correspond to the color of our fragment (perhaps call it `theColor`). Set this variable equal to the vertex position in Object Space (e.g. before it gets transformed…which built in variable can we use?)

Over in the fragment shader, set up the same varying variable and now at TODO #13, change gl_FragColor to be equal to the varying that comes in.

Rerun the program and you should see a much more colorful sphere like the one below.



Q4: Why does it look like that?

Awesome, one last piece.

Step 6 – Make It Move

We'll start to see the power of vertex shaders by having only a piece of our sphere move. To the vertex shader!

This new code will go at TODO #14. NOTE that is goes BEFORE TODO #8. We want to modify our vertex in object space before the transformation takes place. And we only want to modify the vertices that have components greater than zero (that is, X Y & Z are all positive). If that is true, then we will modify our vertex based on time. To know the time, we'll need to pass that from our C++ OpenGL code using a uniform. Set up a uniform of type float called time. We'll now use the following equation to modify our vertex:

$$v' = v + 1.2 * \frac{sin(t) + 1}{2} - 0.2$$

This will add some value based on the sine of time to our vertex. The equation scales the result of sine being in the [-1, 1] range to the [-0.2, 1] range.

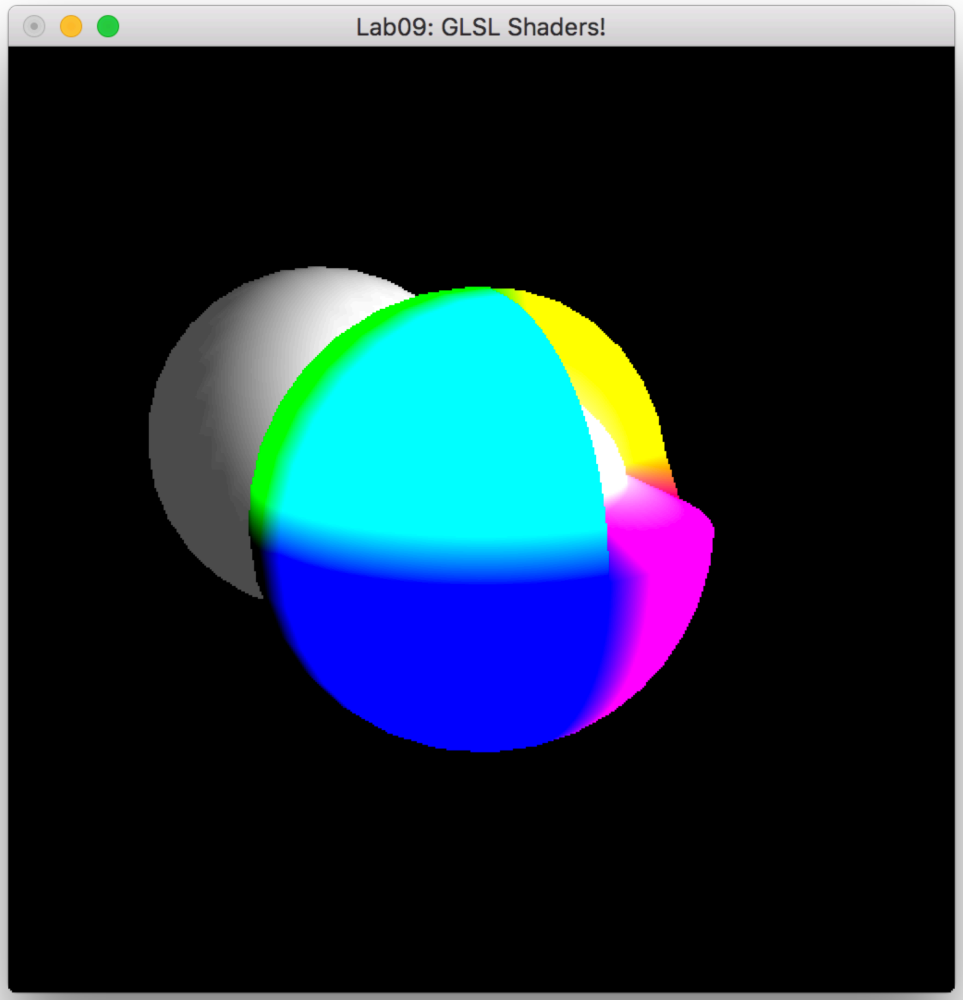
Our modified vertex will then be transformed through the MVP matrix.

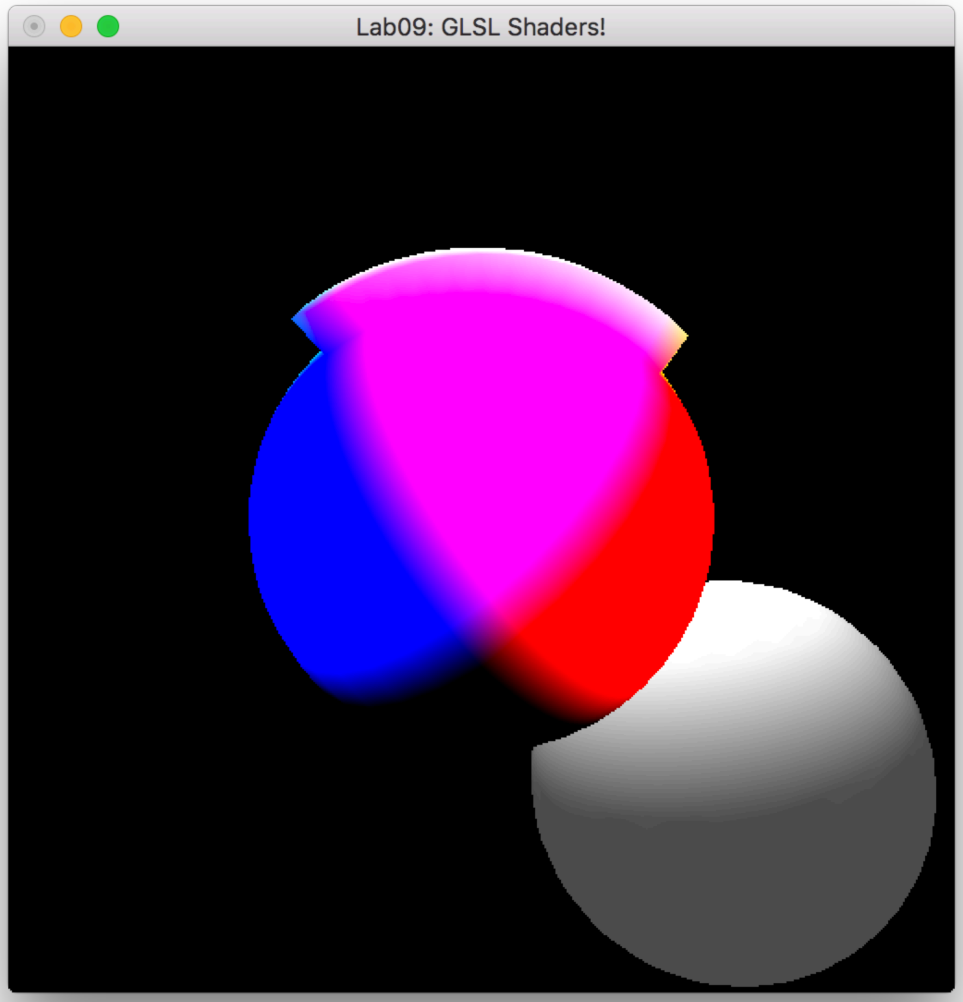
Now we need to set up our uniform variables for this shader. This will go in main.cpp at TODO #15. First, let's tell OpenGL to use our shader. We need to find the location in memory of our uniform. We'll use the function `glGetUniformLocation()`. It takes two parameters, the handle of our shader program and the name of our uniform variable. This name must match exactly the name of the uniform variable we set up in our shader program, the fragment shader to be specific. This function returns a GLuint which is then the location of that uniform. Get the uniform location for the grass texture handle you created. Store this in the global variable created for you.

We will now set the uniform value. To set a value for a uniform, the function in this case is `glUniform1f()`. There are other forms of this function, just as there is with `glVertex*()`. We will pass the elapsed time to the uniform function.

At TODO #16, which must be after we tell OpenGL to use our shader program and before we start drawing primitives, we will ask GLUT to tell us how long our program has been running. We can use the function call `glutGet(GLUT_ELAPSED_TIME)` to get how long our program has been running. This returns the number of milliseconds elapsed, so divide this value by 1000 and store the result in a float. Now pass this value to our shader using the `glUniform1f()` function.

Compile and run. If everything went smoothly, you should see one quadrant of your sphere pulsing in and out. The following two images should help you know what to expect on the two extreme ends.





Q5: Did this lab clear up the confusion involving GLSL/GLEW and shaders? If not, what confusion remains?
Q6: Was this lab fun? 1-10 (1 least fun, 10 most fun)
Q7: How was the write-up for the lab? Too much hand holding? Too thorough? Too vague? Just right?
Q8: How long did this lab take you?
Q9: Any other comments?

To submit this lab, zip together your source code, Makefile, screenshot, and README.txt with questions. Name the zip file <HeroName>_L09.zip. Upload this on to Blackboard under the Lab09 section.

LAB IS DUE BY **FRIDAY NOVEMBER 04 11:59 PM**!!