CSCI 441 - Lab 09
Friday, October 27, 2017

LAB IS DUE BY **FRIDAY NOVEMBER 03 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 and fragment shaders.

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

Step 0 – Get the Libraries copied

There are some new CSCI441 library files included in this lab. Copy them into your existing `Z:/CSCI441/include/CSCI441` folder. These files all end in `*3.hpp`. This means that they will only work with OpenGL 3.0+. We now have all of our cube, sphere, cone, torus, and teapot shapes in a handy VAO/VBO form. Note that this teapot is different. While it accepts a size parameter, the size is ignored and only accepted for compatibility with the OpenGL 2.1 version. This teapot is also a "pure teapot" because it does not have a bottom. There is also a `modelLoader3.hpp` file that will take care of loading in a *.obj file. Suzanne is included for you in the models/ folder.

Step 1 – Set up OpenGL Code

Type make. You may get a warning that we will resolve shortly. Then run the lab by passing in the two shader files.

```
./lab09 shaders/customShader.v.glsl shaders/customShader.f.glsl
```

Hrmm, not quite working. 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 and verify the handle we receive is greater than zero. 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(). NOTE: to pass the actual shader string you need to cast it as* `(const char**)&shaderString`.

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. Again verify the handle we receive is greater than zero. *Hint: use the function glCreateProgram()*.

Next is to attach each shader to the program. Enter these TWO lines at TODO #5A & 5B. *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 #1A and 1B). 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. Don't forget this returns our shader program handle so be sure to set our global variable.

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/customShader.v.glsl`. Note one of our preprocessor directives

```
#version 330 core
```

This states what GLSL version the shader is written in.

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. Unfortunately for us, none of the information we need is readily available to us and we need to pass it all in from the CPU to the GPU. We will use a uniform to pass in the ModelViewProjection matrix. At TODO #A, create a uniform of type mat4 and name it `mvpMatrix`.

Q1: What is a uniform variable for GLSL? Why should the MVP matrix be a uniform?

The MVP Matrix is only half of the equation we need to complete the vertex transformation pipeline. The other half is the vertex itself. At TODO #B, create an attribute as input of type vec3 and name it `vPosition`. We are storing the vertex as a vec3 because all of our vertex positions are points and w = 1. We will hard code the w value in to our shader and not store it in the vertex arrays to save space.

Q2: What is an attribute variable for GLSL? Why should the vertex position be an attribute?

Under Vertex Calculations at TODO #C, 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/customShader.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.

At TODO #D, we first need to create an output to correspond to the color of the fragment. Create an output of type vec4 named `fragColorOut`. We now need to set our output variable. Let's set it equal to white, (1, 1, 1, 1) at TODO #E.

And we're done! Let's head back to the comfort of C++ land to see if this compiles to hook up our data. Open `main.cpp` again.

Step 4 – Hook up the Shader

There are still a few connections that need to be made. After we compile our shader, we need to find the entry points or locations that the inputs are stored at. There are already global variables in place so use the corresponding names.

We first need to get the location of our ModelViewProjection matrix uniform. Place this at TODO #8A and verify this value is non-negative. *Hint: use the function glGetUniformLocation(). The second parameter needs to match the name of the uniform in your shader program exactly. So in our case we'd pass in mvpMatrix.*

Next we need to get the location of our vertex position attribute. Place this at TODO #8B and verify this value is also non-negative. *Hint: use the function glGetAttribLocation(). The second parameter needs to match the name of the attribute in your shader program exactly. So in our case we'd pass in vPosition.*

Where does the attribute location get used? Well, we would pass it to our VBO to say where the vertex position data is stored within our array. This was the step we handwaved over last week when we set up the VBO. If you go to LOOKHERE #3, we are passing this location to our objects library for when the 3D objects get drawn. *NOTE for the library: you need to call this function CSCI441::setVertexAttributeLocations() every time to use a new shader program and want to draw 3D objects with this shader program. The function does accept up to*

three arguments. In order: vertex position location, vertex normal location, vertex texture coordinate location.

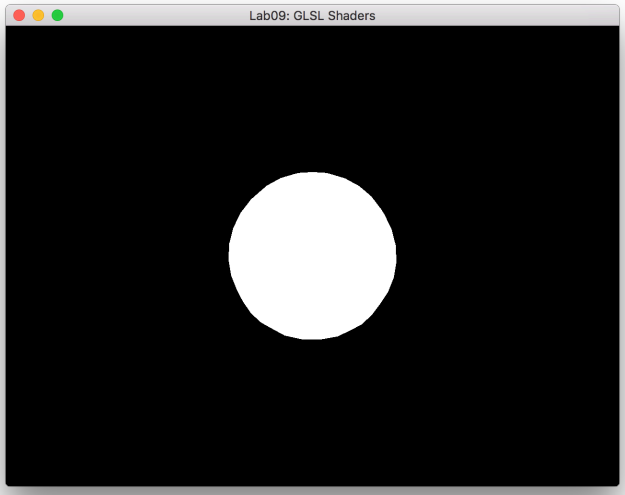
Alright, we're almost there! Now we get to rendering time! At TODO #9A, we must first state what shader program we are using. *Hint: use the function glUseProgram().*

Now we need to send the MVP matrix over to the GPU for our shader program to reference. We will use the function `glUniform*()` for this. It takes many forms and which to use depends on the type of data being sent. Since the MVP Matrix is a 4x4 matrix, we will use the `glUniformMatrix4fv()` version. This function accepts four arguments:

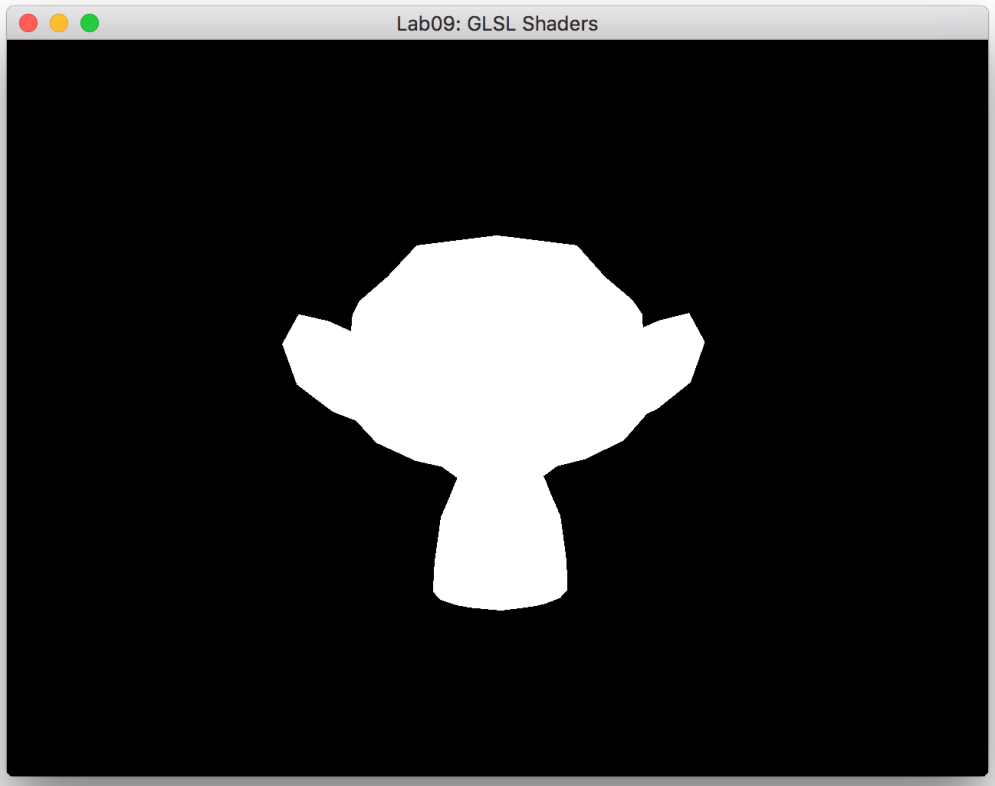
1. The uniform location
2. The number of matrices we are sending (just `1`)
3. If the matrix should be transposed (no – `GL_FALSE`)
4. A pointer to the start of the data. This works like we did when we multiplied our matrices before. `&mvpMtx[0][0]`

And now the moment of truth. Compile and run 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.

If you are seeing a sphere like below



Hooray! It's working! Pat yourself on the back and get ready to modify that shader program. But first, press 7 to look at Suzanne.



Suzanne is somewhat said since she has no face. Let's give her some color so we can see all the details.

Step 5 – Pretty Our Shader Up

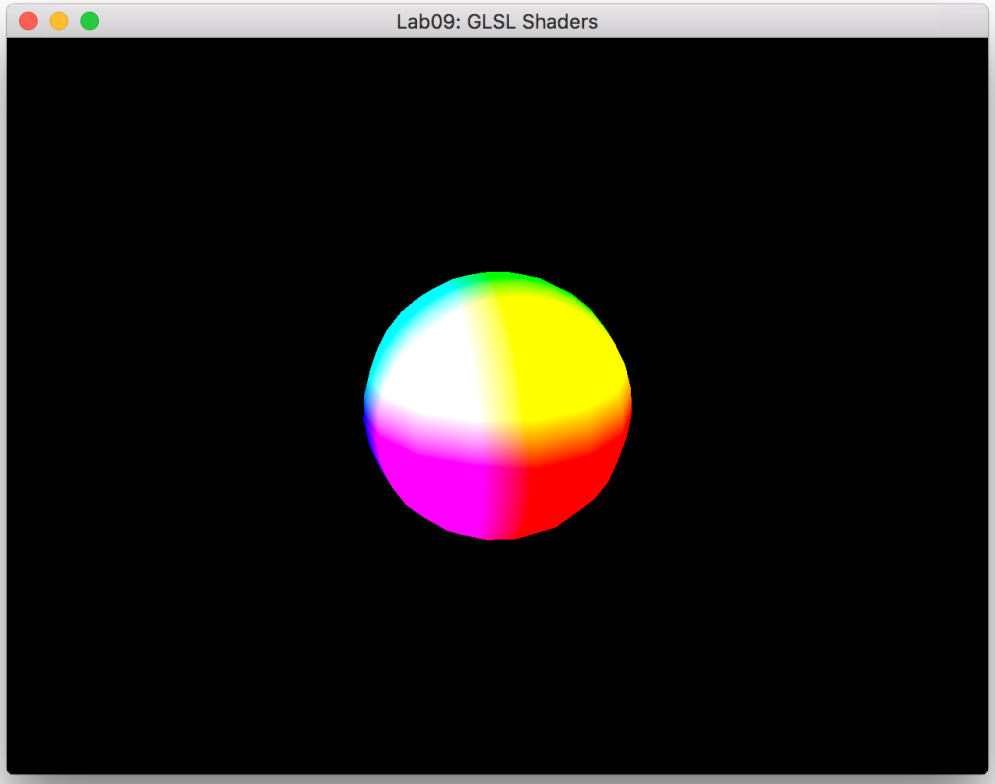
In our Fragment Shader, we don't want to set every fragment just to white. Instead, we'll have it based off of the position of our vertex. Back to the Vertex Shader!

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 of type vec3 at TODO #F1 that will correspond to the color of our fragment (perhaps call it `theColor`). At TODO #F2, Set this variable equal to the vertex position in Object Space (e.g. before it gets transformed…which variable from our vertex shader should we use?)

Over in the fragment shader, set up the same varying variable at TODO #F3 (make sure the name and type match what is stated at TODO #F1!) and now at TODO #F4, change `fragColorOut` to be equal to the varying that comes in. Make sure to properly convert it from a vec3 to vec4. What should alpha be?

Q3: What is a varying variable for GLSL? Why should the color be varying?

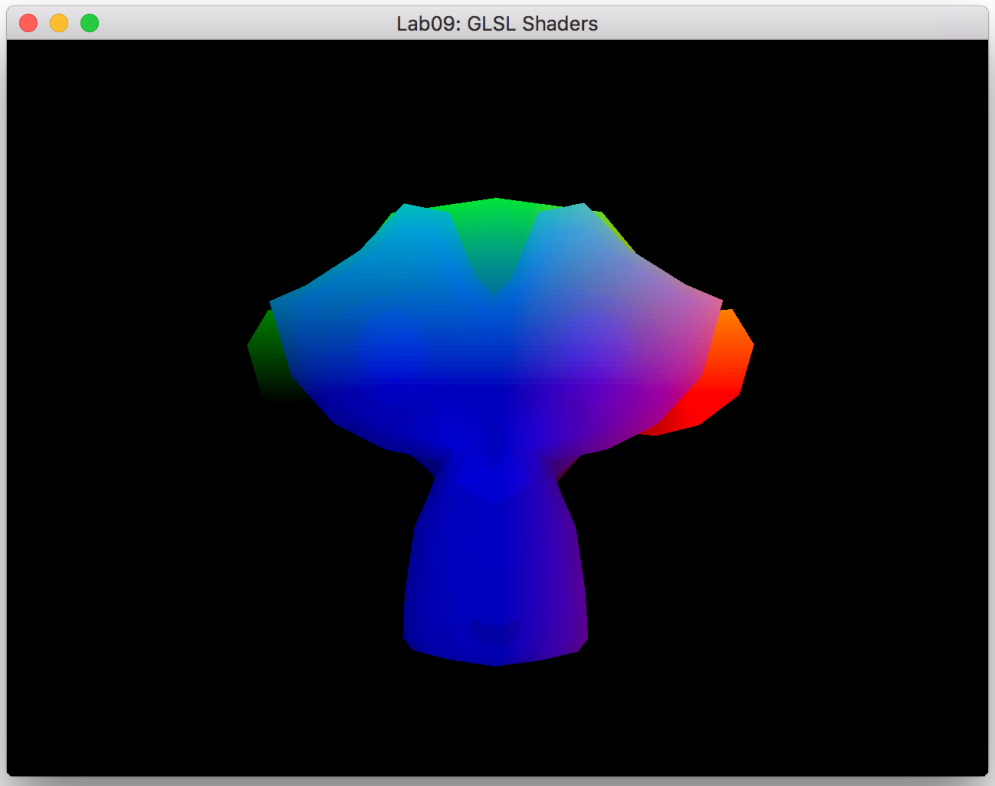
Rerun the program and you should see a much more colorful sphere like the one below.



Q4: 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?

Q5: Why does it look like that?

And Suzanne? Well, she's now smiling and she's also very proud of you.



Awesome, one last piece. Let's make these models dance!

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 #G. We want to modify our vertices 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 at TODO #G1. 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.

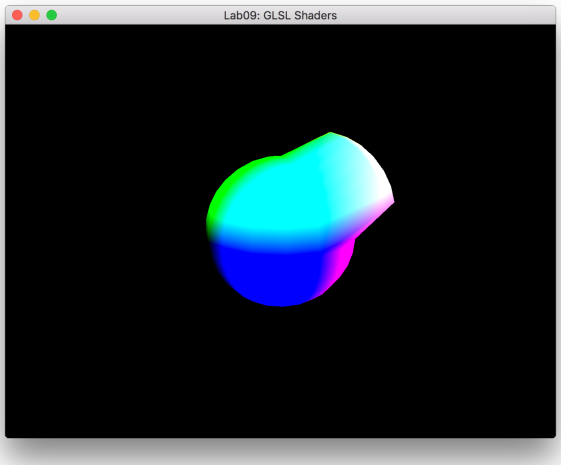
The following steps will go at TODO #G2. NOTE that is goes BEFORE TODO #C. We want to modify our vertex in object space before the transformation takes place. And we only want to modify the vertices that have components greater than zero (that is, X Y, & Z are all positive). If that is true, then we will modify our vertex based on time. Use the above equation to compute the new vertex position. To save you a compile, see an error, fix it step – we cannot modify our `vPosition` attribute and use that in the equation. `vPosition` comes in to the shader as a constant. We'll need to make a new variable (call it `newVertex`), use that as v' in the equation above, and then put `newVertex` into our transformation equation at TODO #C.

Now we need to pass in time to this shader. Back to `main.cpp` one last time! This will go in `main.cpp` at TODO #10A. We'll use the function `glGetUniformLocation()`. Get the uniform location for the time variable you just created.

We will now set the uniform value. Since this uniform corresponds to a single float value, the function we will use is `glUniform1f()`.

At TODO #10B, we'll pass the time to the shader program. We can use the function call `glfwGetTime()` to get how long our program has been running. This returns the number of seconds elapsed and the resolution is system dependent (usually micro- or nanoseconds). 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 image should help you know what to expect on the extreme end.



And Suzanne? Well, she's not sure what to think now.

You can use this lab to test future shaders now that you know it is working. As long as your shader program has a `vPosition` attribute and `mvpMatrix` uniform, your OpenGL program will interface with the shader.

For the assignment, the vertex shader won't be moving vertex positions. You then need to implement the Blinn-Phong Illumination Model as well as Texturing. This will require additional uniforms, attributes, varyings, and calculations in your shader program(s).

Q6: Did this lab clear up the confusion involving GLSL/GLEW and shaders? If not, what confusion remains?
Q7: Was this lab fun? 1-10 (1 least fun, 10 most fun)
Q8: How was the write-up for the lab? Too much hand holding? Too thorough? Too vague? Just right?
Q9: How long did this lab take you?
Q10: 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 L09 section.

LAB IS DUE BY **FRIDAY NOVEMBER 03 11:59 PM**!!