

CSCI 441 - Lab 04
Friday, September 23, 2022
LAB IS DUE BY **FRIDAY SEPTEMBER 30 2022 11:59 PM!!**

Today, we'll look at how to properly use shaders. We will look at a simple example that does the minimum FFP requirements and then do a slight modification to the vertex and fragment shaders. You will find the Shaders Checklist helpful to have on hand as you work through the steps.

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

Step 0 – Get the Libraries copied and look around

There are some new CSCI441 library files included in this lab. Copy them into your existing `Z:/CSCI441/include/CSCI441` folder. There is a `ModelLoader.hpp` file that will take care of loading in a `*.obj`, `*.off`, `*.stl`, or `*.ply` file. Suzanne is included for you in the `models/` folder as an obj file. There are some other dependent files included as well such as the completed Fixed and Free camera implementations.

Take a look at how our engine is set up, it's a little different and a bit more efficient. The implementations for our event callbacks pass the relevant event information through to the engine. We are now handling the event inside the engine. Inside the scope of the engine we can access all members directly so we have a much smaller and cleaner interface with our engine. We can keep the logic full internal to the engine.

Step 1 – Set up OpenGL Code

Before running your program, be sure to set the Run > Edit Configurations > Working Directory to be the parent .. so CLion will be able to find all of your model and shader files.

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

- `Lab04Engine.hpp`
- `Lab04Engine.cpp`
- `shaders/customShader.v.glsl`
- `shaders/customShader.f.glsl`

Our goal is to render a set of models with 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.

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. Near the bottom of the `Lab04Engine.cpp` file is the implementation for `_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 #01`. *Hint: use the function `glCreateShader()`.*

Reading the shader from file to a string is handled for you at `LOOK HERE #1`. The next step is to now send our code to the GPU. Do this at `TODO #02`. *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 #03`. *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 #04`. 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 #05`. *Hint: use the function `glAttachShader()`.*

Lastly, tell the GPU to link our shader program together at `TODO #06`. *Hint: use the function `glLinkProgram()`.*

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

Once we are done creating the program, we are good little programmers and will clean up our memory. First, at `TODO #07`, detach the vertex and fragment shaders from your shader program. *Hint: use the function `glDetachShader()`.*

Second, at `TODO #08`, delete the vertex and fragment shaders from GPU memory. *Hint: use the function `glDeleteShader()`.*

In `_setupShaders()` is where we will call our shader helper functions. At `TODO #09A/B`, call our `_createShaderProgram()` function passing in the filenames of our two shader files. Don't forget this returns our shader program handle so use a data member for it. (*Note: read the comment in the code at `TODO #09A`*)

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 410 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 ourselves. 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 our vertex positions are points and $w = 1$. We will hard code the `w` value into 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. We can also do an early fragment discard prior to any of our per-fragment tests, but again we won't 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 the engine files 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 data members in place so look at the structs in the header file and use the corresponding names.

We first need to get the location of our `mvpMatrix` uniform. Place this at `TODO #10A` and verify this value is non-negative. *Hint: use the function `glGetUniformLocation()` and our ready made variable at LOOK HERE #2. 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 #10B` and verify this value is also non-negative. *Hint: use the function `glGetAttribLocation()` and the ready made variable. 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 `LOOK HERE #3 & 4`, we are passing this location to our objects and model 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 #11`, we must first state what shader program we are using. *Hint: use the function `glUseProgram()`.*

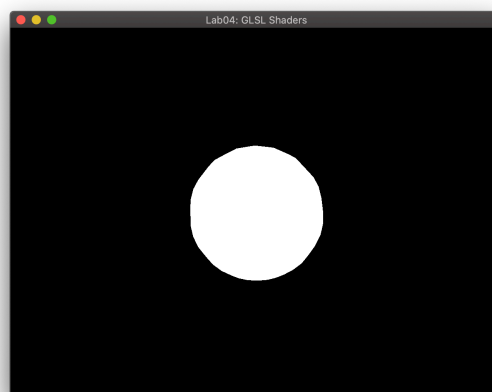
Now we need to send the MVP matrix over to the GPU for our shader program to reference. We will use the function `glProgramUniform*()` for this. It takes many forms and which to use depends on the type of data being sent. Since the MVP Matrix is a 4x4 matrix, we will use the `glProgramUniformMatrix4fv()` version. This function accepts five arguments:

1. The shader program handle
2. The uniform location
3. The number of matrices we are sending (just 1)
4. If the matrix should be transposed (no - `GL_FALSE`)
5. A pointer to the start of the data. For matrices, this is `&mvpMtx[0][0]`

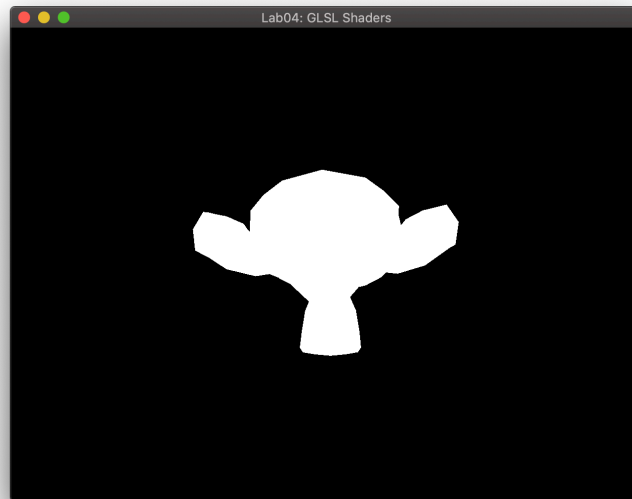
And now the moment of truth. Compile and run your program

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

If you want to fly around the scene, this lab contains a working free cam implementation. Pressing space will fly you forward. You can click and drag the mouse to look around. Or you can use the WASD keys to also look around. Pressing shift will fly you backwards when also pressing space.

Step 5 – Pretty Our Shader Up

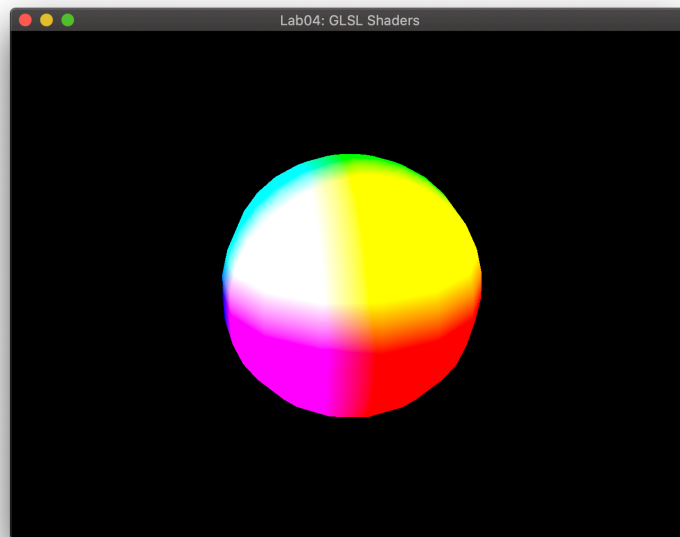
In our Fragment Shader, we don't want to set every fragment just to white. Instead, we'll have it based off of the position of our vertex. Back to the Vertex Shader!

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?

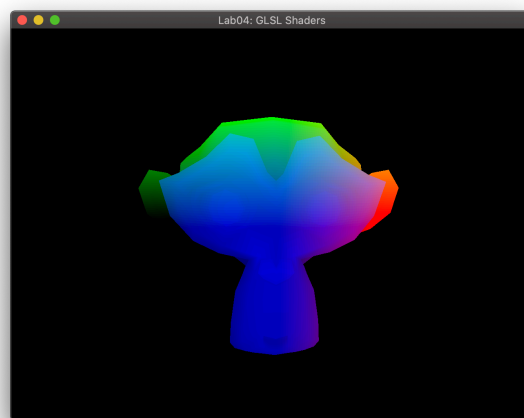
Rerun the program and you should see a much more colorful sphere like the one below.



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.

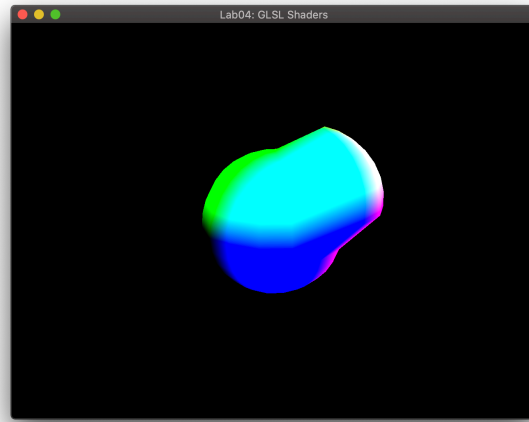
The following steps will go at `TODO #G2`. **NOTE** that it goes **BEFORE** `TODO #C`. We want to modify our vertex in object space before the transformation takes place. And we only want to modify the vertices that have components greater than zero (that is, X Y, & Z are all positive). If that is true, 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 the engine one last time! This will go at `TODO #12A/B/C`. 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 `glProgramUniform1f()`.

At `TODO #12C`, 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 `glProgramUniform1f()` 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.

The last step has already been done for you. At `LOOK_HERE #4`, we are deleting the shader program from the GPU to free the memory.

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.

Q6: Did this lab clear up the confusion involving GLSL 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, screenshot, and README.txt with questions. Name the zip file `<HeroName>_L04.zip`. Upload this on to Canvas under the L04 section.

LAB IS DUE BY **FRIDAY SEPTEMBER 30 2022 11:59 PM!!**