CSCI 444/544 - Lab 2
Friday, January 29, 2021

LAB IS DUE BY **FRIDAY FEBRUARY 05 11:59 PM**!!

This lab will get you working with shader subroutines. We'll put one in to the Vertex Shader and one in to the Fragment Shader.

Run the program and you should see the program start up! Yay, a rotating cube above a ground plane. For now, ignore the error about not finding the time uniform. We will be adding it in later.

Step 0 – Patch Two Library Files

In the include/CSCI441 folder is two updated course library files. Copy these to your system ~/include folder.

Step 1 – Changing the Colors

If we look at `colorPassThrough.f.glsl` we will see that it is setting the output color to be the input color. We are going to add some new coloring options. We'll be adding this code under the subroutines comment at `// TODO A`.

First, define a subroutine function prototype named processColor that takes a vec3 as input and returns a vec3 as output.

```
subroutine vec3 processColor(vec3)
```

Next, we need to create the subroutine uniform to control which subroutine to use.

```
subroutine uniform processColor colorProcessor;
```

Recall that this creates our function pointer. Later, when we need to call the implementation, we will call the function colorProcessor(vec3) and the corresponding implementation will be called at runtime.

Now we need to actually make a subroutine to use. Create a function called passThrough that simply returns the parameter. Make this a subroutine of processColor.

```
subroutine(processColor)
vec3 passThrough(vec3 inputColor) {
    return inputColor;
}
```

Now down at `// TODO B`, we need to call our function! We want to call our function pointer here.

~~`fragColorOut = vec4( color, 1 );`~~

```
fragColorOut = vec4( colorProcessor(color), 1 );
```

At this point, we can just rerun our program and we should see the same output as before. Take a look at the shader output in the console. You will note that there is now information being displayed about the subroutine you just created. Let's add a second subroutine and look at the difference.

Create another subroutine function for processColor, we'll call it `grayScale`. Place this below the definition for passThrough. This function should average the input color. First sum the three channels (r, g, b) and then divide the total by 3.0. Return a vec3 that has each component equal to the average value.

Rerun the program. Your cube and ground will still be colored. But looking at the terminal output you'll see a second line under the Fragment Shader subroutines. `passThrough` is listed first in the code and has an index of 0, `grayScale` is listed second and has an index of 1. By default, the subroutine with index 0 is used.

In the Fragment Shader code, move the `grayScale` function to be above the `passThrough` function. Rerun your program. Everything should have gone gray! And the shader info reflects the ordering change as well.

We can manually change the effects of the shader by reordering the code. But we would much rather allow the user at runtime to toggle between the different functions. Jump over to `main.cpp` and we'll add some OpenGL code now.

We need to perform a few steps

1. We need to get the indices of our subroutines
2. We need to set which subroutine to use

Go to `// TODO #1` first. We need to create a `struct`, similar to the uniform and attribute locations, for our indices. Name the `struct ColorShaderSubroutineIndices`. Give it three variables of type `GLuint` and name them `fragment`, `fPassThrough`, & `fGrayScale`. The first will be used to keep track of which subroutine we are using. The latter two will hold the index of the corresponding subroutine. Finally, create an object of the struct type named `colorShaderSubroutineIndices`.

Now down to `// TODO #2` inside of `setupShaders()`. We need to query the subroutine indices for each function. There are a series of OpenGL calls to do, but these have been simplified into the helper class for you. Call the function

```
shaderShaderProgram->getSubroutineIndex( shaderType, "functionName" )
```

to get the index for a subroutine. The *`shaderType`* will be one of `GL_VERTEX_SHADER` or `GL_FRAGMENT_SHADER`. The *`functionName`* is then a string of the corresponding function, in our case `passThrough` and `grayScale`.

Set the `fPassThrough` and `fGrayScale` properties equal to the subroutine index from the function call above.

Lastly, set the `fragment` property to be equal to `fPassThrough` by default.

Getting closer! Every time we change which program we are using, all state information for the subroutines is lost. Down in renderScene(), we need to set the subroutine for each object. The function call is

```
glUniformSubroutinesuiv( shaderType, n, indexArray );
```

`shaderType` is the same as above. *`n`* is the number of subroutines, in our case just `1`. The *`indexArray`* is then a pointer to the list of indices.

At `TODO #3`, we want to apply the pass through subroutine to the ground. Pass in the last value as `&( colorShaderSubroutineIndices.fPassThrough )`.

Then at `TODO #4`, we will pass in the current subroutine when the cube is drawn. Pass in `&( colorShaderSubroutineIndices.fragment )`.

At this point, we can compile our program and run. We should see the colors back. But we can't change it yet. One last step.

Go up to `// TODO #5` in the `key_callback` method. We now need to toggle these values. If the user presses the `1` key, then set the `fragment` property to be equal to `fPassThrough`. If the user presses the `2` key, then set the `fragment` property to be equal to `fGrayScale`.

Compile, run, press 1 and 2. The colors change!

Perform the above steps again to add a third option for the fragment shader. Create another subroutine function called `timeFlow` that does the following:

```
vec3 timeFlow(vec3 inputColor) {
  return inputColor * vec3( (sin(time)+1.0)/2.0,
                            (cos(time)+1.0)/2.0,
                            (cos(time)+1.0)/4.0 + (sin(time)+1.0)/4.0
                          );
}
```

The user should be able to press `3` to switch to this coloring scheme.

Step 2 – Move It

Time to repeat all of the steps on your own now in the Vertex Shader. Create a subroutine to process the vertex. The goal is to change `// TODO D` to the following.

~~`gl_Position = mvpMtx * vec4(vPos, 1);`~~
`gl_Position = mvpMtx * vec4( vertexProcessor(vPos), 1 );`

We'll want one subroutine of `vertexProcessor` to be a `passThrough`, that simply returns the `inputVertex`.

Create a second subroutine called `timeFlow` that performs the modification to our vertex:

```
vec3 timeFlow(vec3 inputVertex) {
  return inputVertex + vec3( cos(inputVertex.x + time),
                             sin(inputVertex.z + time),
                             0
                           );
}
```

Create a third subroutine of your choosing.

The `4, 5, and 6` keys should toggle these three options. You'll need to go through `TODO`s `C` and `D` in the vertex shader and then modify `TODO`s `1-5` in `main.cpp`.

Step 3 – Moving Forward

At this point, your task for Lab2 is done. You can now incorporate this in to A1 to toggle between Phong Specular Illumination and Blinn-Phong Specular Illumination.

Submit your entire project as a single zip file to Canvas.

LAB IS DUE BY **FRIDAY FEBRUARY 05 11:59 PM**!!