COSC 4370 Homework 3

Name: Jordan Foster | **PSID:** 1669900
March, 2022

1. Assignment

The goal of this assignment is to recreate a 3D cube, utilizing the Phong shading technique.

2.1 Implementation for Phong.vs

The *phong.vs* file contained several variables, including the 4x4 matrices *model*, *view*, and *projection,* along with the 3D vectors *Normal* and *FragPos.* The transformation matrices serve to produce coordinates and transformations to provide the necessary components to produce our perspective projection for the scene in question. The 3D vectors serve to be used in tandem (using the perpendicular angle from the given vertices) to determine world position within the scene.

To properly set the shaders, we first look at the most obvious variable: *gl_Position.* Given the fact that the pipeline for a scene is *model —> view —> projection*[1], it can be concluded that *gl_Position (*which serves to position the camera for the scene within the clip space) is to be equal to projection * model * view, since the transformations are read from right to left[2]. It is then multiplied by the 4D vector *positionLayout* variable (renamed from *position*) to give it functionality.

FragPos and *Normal* were more straightforward. Since *FragPos* is for the positions of the fragments within clip space, the initial and correct assumption was that it could be set to the product of the model matrix and the 4D vector *positionLayout* variable. For the *Normal* vector, it was apparent from the name that it represented the normal transform. Given that the normal transform is the transpose inverse of the model matrix, it was initially assumed that Normal could be set equal to *mat3(transpose(inverse(model))*normalLayout,* creating the desired 3D matrix. However, the model matrix is orthogonal since our values for the scene are normalized[3], therefore each vector within the matrix is perpendicular to each other on the surface of the object. Because it is orthogonal, the transpose inverse of the matrix would be redundant as result

[1] Reference: https://jsantell.com/model-view-projection/

[2] Reference: https://learnopengl.com/Getting-started/Coordinate-Systems

[3] Reference: https://en.wikipedia.org/wiki/Orthogonal_matrix

in the original *model* matrix[4]. Therefore it can just simply be set equal to *mat3(model)*normalLayout.* With that completed, the *phong.vs* fixes are done.

2.2 Implementation for Phong.frag[5]

Now we focus our attention to the fragments for the shaders. Given that the Phong technique utilizes the combination of ambient and diffuse lighting, it can be predetermined that we must define both ambient and diffuse lighting within this file.

We can start with normalizing the *Normal* vector to be used as the *norm* vector within this file. From there we have the ambient vector be the product of a small float (0.1f in this case) and the scene's *lightColor* to represent that the strength of the lightning is dull. With lighting in mind, we determine the *lightDir* (light direction) by normalizing the difference between the lightPos and FragPos and storing it within a 3D vector. Likewise we can determine the view direction (*viewDir)* by normalizing the difference between the *viewPos* and *FragPos* vectors. The reflection's direction (*reflectDir*) is given by reflecting the negative value of the *lightDir* and our previously made *norm* vector (*reflectDir = reflect(-lightDir, norm))*.

Now, to have a source of light, it would result in a source of "spotlight" effect, known as a specular, the brightness of which is determined by what exponential power it is raised to and a To set this value, we take the dot product of our *viewDir* and *reflectDir* vectors, and raise that value to the power of 64. To add this spotlight to the object, we create the specular as a 3D vector that is the product of the dot product we just calculated and the *lightColor.* With that completed, all that is left is to calculate the result. As mentioned earlier, the Phong technique is a combination of ambient and diffuse lightning—additionally, since we are adding a specular, it is also part of the result. So, *result = (ambient + diffuse + specular) * objectColor* to create the proper color, lightning, and shading for our 3D object. With that, the *Phong.frag* fixes are complete.

2.3 Implementation for Camera.h

The completion of the *Camera.h* file was rather trivial, given the knowledge of how a view matrix is to be created. The view matrix is being created with the *glm::lookAt* function, which uses 3 axes—camera position, camera up, and camera front[6]. These variables were provided in the source code, so the implementation was straightforward: v*iew = glm::lookAt(this->Position,*

---

[4] Reference: https://stackoverflow.com/questions/53192459/glsl-normal-vector-transformation

[5] Reference for this section is all sourced from: https://learnopengl.com/Lighting/Basic-Lighting

[6] Reference: https://learnopengl.com/Getting-started/Camera

this->Position + this->Front, this->WorldUp), which is then returned from the function. With that, the view matrix is completed and the *Camera.h* fixes are complete.

2.4 Implementation for Main.cpp

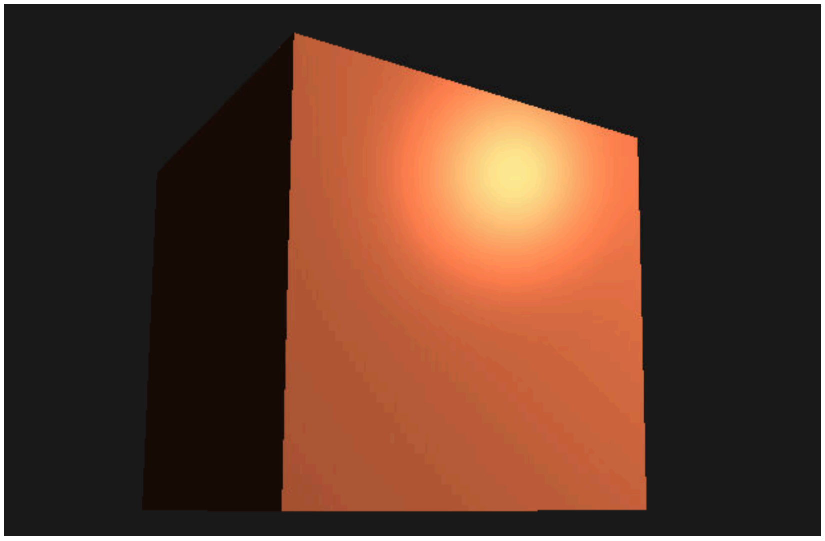
Lastly, the final fix to complete the scene is to initialize the perspective projection matrix. For a perspective projection, an angle is needed to showcase the depth-dependent effects within a 3D scene. Fortunately, there is a *glm::perspectiv*e function to be utilized, that allows us to plug in a desired angle (in this instance, 45.0f radians), the quotient of the width and height, and two variables to determine the near and far clipping planes[7] and produces a 4x4 matrix for the perspective projection. As such, we can plug in some values and produce:

glm::mat4 projection = glm::perspective(glm::radians(45.0f), (float)w/(float)h, 0.1f, 100.0f)[8].

With this completed, the perspective projection matrix is completed, the *main.cpp* fixes are complete, as well as the shader fixes. As such, our assignment is complete.

3. Result

With all fixes in place, we are able to produce our shaded square. Using the WASD, the camera position is maneuvered to reveal that the square is indeed a 3-dimensional cube with the Phong technique properly implemented and the desired scene is produced.



[7] Reference: https://learnopengl.com/Getting-started/Coordinate-Systems

[8] Sourced from: https://learnopengl.com/Getting-started/Coordinate-Systems