msihapanya Final Project Reflection CS-330

I mainly adapted from the tutorial code that was available to us. First, I focused on constructing the cube. Then, for the plane attached to the cube to make a lens wipe box, I thought of the plane as two triangles. I placed down the four vertices and then marked the indices to draw two triangles, making the final plane. Although I chose a cylinder and a sphere to represent a mug and a stress ball, I was not able to integrate the example code into my ongoing milestone. Placing the sphere code caused only a quarter of a circle to render. Instead, I chose to place down a pyramid in place of the stress ball, and an open prism to represent the mug. In creating the pyramid, I transferred what I learned from the pyramid assignment, and used the texture I would have used on the stress ball. For the mug, I constructed a simple cube first. Then, I deleted the top face and lengthened the y positions to resemble the height of a mug. I also widened the top vertices to simulate a mug shape. For each milestone, I spent a lot of time figuring out how to do each piece – the triangle rendering, texturing, and lighting. During texturing, my program crashed because my channels were incompatible with my image. It took much debugging to resolve that. And while I added light, the texture would disappear from the cube. I figured out it was because of my stride code and how I was enabling my vertex attribute pointers for each object. But in trying to render a sphere or a cylinder, I did not understand the different requirements for these object’s structures. While I was able to create a 3D scene, I believe I failed in creating other primitive objects because of my lack of modularization in my code. If I understood how to separate the VAOs/VBOs, shaders, rendering calls, transformation matrices, and uniforms into different files for reuse, I believe I could’ve successfully rendered other primitive shapes.

The user navigates the 3D scene through the UMousePositionCallback(), UMouseScrollCallback(), and UProcessInput() functions. These functions take in the mouse and keyboard as input devices. The first function tracks the x and y position of the mouse, then uses the ProcessMouseMovement() method from camera.h to capture any mouse pitch and yaw from the user. The second function allows the user to adjust the speed of the camera with the mouse scroll wheel. ProcessMouseScroll() tracks the camera’s movement speed and changes it depending on the input received from the mouse scroll. Scrolling up moves the camera faster, and scrolling down moves the camera slower. The third function handles keyboard input. Multiple conditional statements exist for the W (up), A (down), S (right), D (left), Q (up), E (down), and P keys. When W, A, S, D, Q, and E are pressed, ProcessKeyboard() is called and the movement speed is multiplied by time to create the user’s velocity. Pressing P toggles an orthographic projection of the scene, which is ensured in the UProcessInput() function and in the transformation code with glOrtho(). Using all these inputs together, the user can move around the scene and adjust the camera accordingly.

Unfortunately, I failed to totally modularize this project. I spent a lot of time debugging my milestones for their basic functionalities. While coding the milestones as I followed the tutorials, I looked at MultipleObjectsExample and didn’t understand how to convert the singular source file I’ve been working on, into multiple files with reusable functions. So I kept focusing on developing at least the objects and having the scene constructed. It was confusing as the example loaded the vertex and fragment shaders as separate files as opposed to coding them in line with the solution. The only reusable functions I was able to implement were the ones given from the tutorials and were then further modified as I worked on the milestones. UCreateShaderProgram() allowed me to create a shader program for each object with its shaders and program ID. UCreateMesh() is just a singular function and encompasses all objects, but I now see that I could’ve created a class for each object which holds its own mesh inside. URender() similarly could’ve been split for each object, to the effect of how MultipleObjectsExample created sphere and cylinder objects and called S.Draw() and C.render().