Design Decision and Reflection

A Triangle & Cube Studios client would like to replicate a two-dimensional image with a three-dimensional model. I have developed code to reproduce a three-dimensional model of the two-dimensional image utilizing the OpenGL API and GLSL(OpenGL Shading Language) libraries. The image contains four objects a coffee mug, calculator, vitamin bottle, and Eau de Toilette. The code renders objects in three dimensions, applies realistic textures, and implements user navigation controls. Users can navigate the scene with the mouse cursor, mouse scroll, and keyboard inputs. The ‘A’ key to move forward, ‘S’ to move backward, ‘A’ to move left, ‘D’ to move right, and the mouse cursor to look around the scene. The mouse scroll up increases the camera speed, mouse scroll down decreases the camera speed.

The three-dimensional representation utilizes low-polygon models made with primitive shapes and composed of many triangles connected by their vertices. The four objects in the image have bodies that resemble basic shapes. For example, the calculator is an elongated cube, the coffee mug is a cylinder, and Eau de Toilette is a cube with a cylinder stacked on top of it. I chose the four objects in the image based on the requirement that the representation utilizes primitive shapes and the characteristics of the objects’ bodies. I began to think of how to replicate various primitive shapes with triangles. I knew I would have to develop functions to generate vertex data because the number of triangles and vertices required to form different shapes would be substantial. Hard coding the vertices would have been time-consuming and inefficient. If there were an option to utilize .obj files, I would have selected more complex objects for the three-dimensional representation.

The completed program is modular and consists of different class components. The class components for the program are Window, Mesh, Shader, Texture, Shape, Cube, and Cylinder. I didn’t realize how much more challenging it would be to modularize the program this way with these components. Creating a single program to render a three-dimensional scene was straightforward and manageable. Global variables, pointers, and structs made it convenient to utilize various functions, libraries, and components within a single program. However, it starts to get tricky once you start modularizing the components and its functions.

The program begins by creating and initializing a Window object. The program’s Window class creates and manages a GLFW window and initializes the GLFW and GLEW libraries. The Window class is also responsible for processing inputs and GLFW callbacks. Creating a Window class component made it extremely challenging to implement the other class components. Processing inputs and utilizing the GLFW callbacks was the most difficult to integrate into the OpenGL program’s main function and entry point. I had to resolve an issue integrating deltaTime and the glfwGetTime() function. Ultimately I created an update function for the Window component that updates the deltaTime and deltaFrame class members. It seems like a simple solution, but at the time, I was in the mood to try the most complicated ideas first. Not having global variables for the camera, deltaTime, and deltaFrame was another challenge in modularizing the Window component. The GLFW callback functions are responsible for processing inputs and updating the camera attributes. However, the GLFW callback functions are static, and according to the Visual Studio C++ compiler, a nonstatic member reference must be relative to a specific object. Long story short, I had to dig through the GLFW documentation to figure out how to cast the program’s main window to a static window object using a glfwGetWindowUserPointer().

The Cylinder and Cube classes inherit functions and attributes from the Shape class. I found the shape class very useful for iterating different types of objects. The shape class also sets and gets common attributes passed to the shader program for different types of shapes. An example, it can set object position and object scale vectors, and then the Mesh’s render function can take in a Shape as its’ parameter and can call shape.getScaleVector() for the glm::scale() and glm::translate() functions.