Design Decision and Reflection

A client of Triangle & Cube Studios has expressed interest in creating a three-dimensional scene of a two-dimensional image. To fulfill this request, I have developed an application using the OpenGL API and GLSL (OpenGL Shading Language) libraries. My code generates a three-dimensional model that replicates the original image with four objects: a coffee mug, a calculator, a vitamin bottle, and Eau de Toilette.

The objects in the model are rendered in three dimensions with realistic textures, and users can navigate the scene using various input methods. The mouse cursor can be used to pan around the objects, while keyboard inputs such as 'W', 'A', 'S' and 'D' can be utilized to move forward, left, backward, and right, respectively. The camera speed can also be adjusted using the mouse scroll.

To create the low-polygon models, I utilized primitive shapes composed of multiple triangles. Each object was designed to resemble fundamental shapes, such as the elongated cube for the calculator, cylinder for the coffee mug, and a cylinder stacked on top of a cube for Eau de Toilette. These shapes were chosen to conform to the requirement of using primitive shapes and their body features. To create these models, I had to develop functions to generate vertex data, as hard-coding each vertex would be highly time-consuming and inefficient.

While I chose these models to represent four objects in the image, I would have preferred more complex models if given the option of utilizing .obj files. However, as the models needed to be composed of primitive shapes, I believe the decision to utilize lower complexity models was appropriate.

The program has been successfully completed utilizing various class components, making it fully modular and portable. These component classes include Window, Mesh, Shader, Texture, Shape, Cube, and Cylinder, each serving a unique purpose in the program. Although, modularizing the program proved to be more challenging than anticipated, especially with these components in place.

Creating a single program for rendering a three-dimensional scene was manageable due to the ability to use global variables, pointers, and structs within one single program. However, modularizing the components' functions highlighted the trickiness of this process. The program starts by initializing the Window object which creates and handles a GLFW window and initializes both the GLFW and GLEW libraries. The Window class also manages the processing of inputs and GLFW callbacks. Developing the Window component became a challenge to implement the other class components. Troubleshooting and integrating the callbacks and inputs into the program's main function presented a significant obstacle.

One issue involved integration of deltaTime and the glfwGetTime() function. Eventually, I established an update function for the Window component, which helped updating the deltaTime and deltaFrame class members. Although it seems simple, this took time to develop as I initially opted for more complex solutions. Another issue was the lack of global variables required for the camera, deltaTime, and deltaFrame during modularizing the Window component. The GLFW callback functions processed input and updated the camera attributes. However, the GLFW callback functions are static, and according to the Visual Studio C++ compiler, a nonstatic member reference must be related to a specific entity. I resolved this by studying the GLFW documentation to determine how to cast the program's main window to a static window object using the 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.