**Development Choices:**

When I began my computer graphics project, my initial design proposal envisioned a simple scene: a table with a cup, a napkin, a plate, and a bowl.



Figure 1: Initial scene proposal

However, as I progressed, I realized the importance of constant visualization and real-world reference in the creative process, which significantly influenced the direction of my project, resulting in a significant departure in ultimate result:

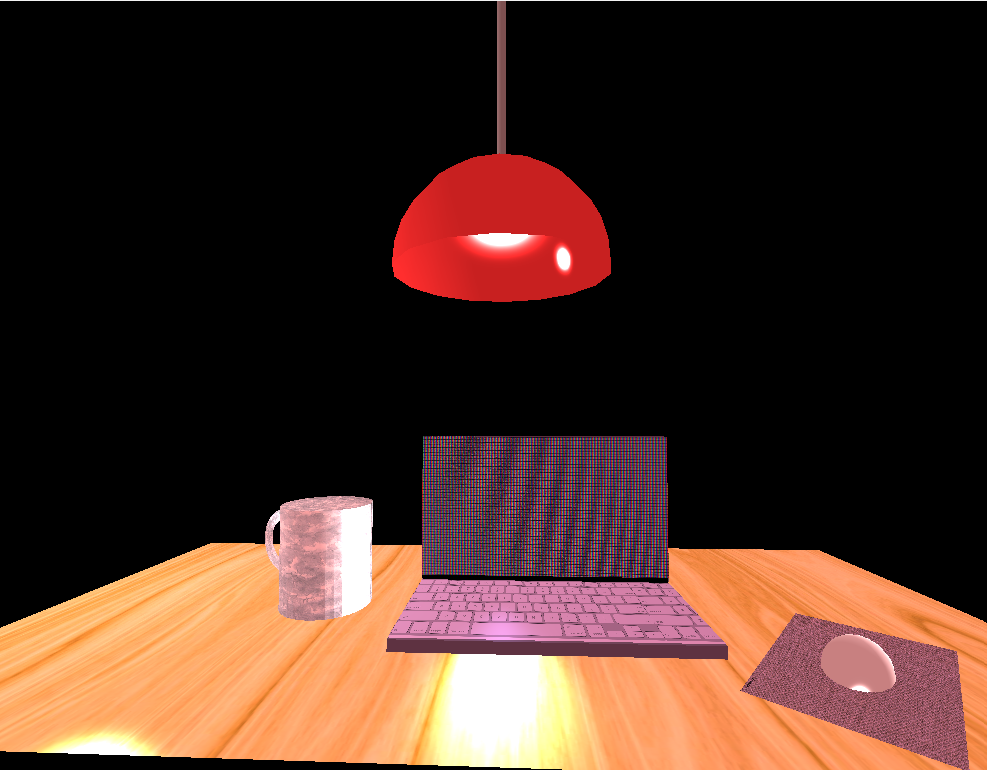


Figure 2: Final scene outcome

I began my project with the cup, adding to its creation over each milestone. As I progressed, I found myself frequently referencing the actual cup on my desk rather than relying on the image from my original design. This habit of looking at physical objects for accuracy made me reconsider the practicality of my initial scene. Consequently, I decided to shift my focus to a more familiar and accessible environment: my work desk. This updated scene included a laptop, a table, a cup, a mouse and mouse mat, and an overhead lamp.

The new design not only better reflected my everyday surroundings which I could actually reference, but also met the project requirements, featuring at least four object, three of which were complex objects:. the cup (composed of a cylinder and half-torus), the laptop (composed of two transformed boxes into rectangles, and plans to hold textures), and lamp (composed of a half sphere and a cylinder). Moreover, having an actual reference allowed me to more easily visualise the lighting in the scene, giving me the direction to recreate the soft red lighting of my lampshade. The light itself added some vertical dynamism that I felt was more appealing to experience in a 3D environment. This experience taught me that adapting designs to leverage available references and familiarity can enhance both creativity and execution in computer graphics projects.

**Navigation:**

The user navigation system in intended to be intuitive and versatile, utilizing multiple input devices: the keyboard, mouse, and the mouse scroll wheel. Keyboard controls, implemented in ProcessKeyboardEvents(), provide a familiar way that any PC gamer would recognise to navigate the scene. Keys W, A, S, and D allow forward, left, backward, and right movement, while Q and E enable vertical adjustments.

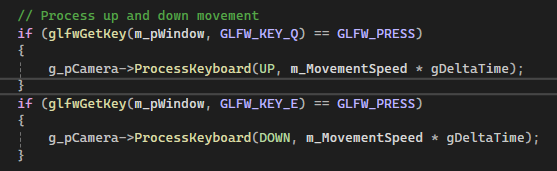
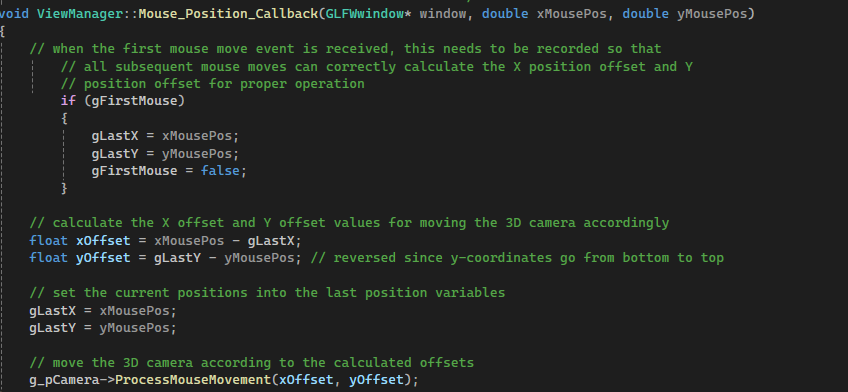


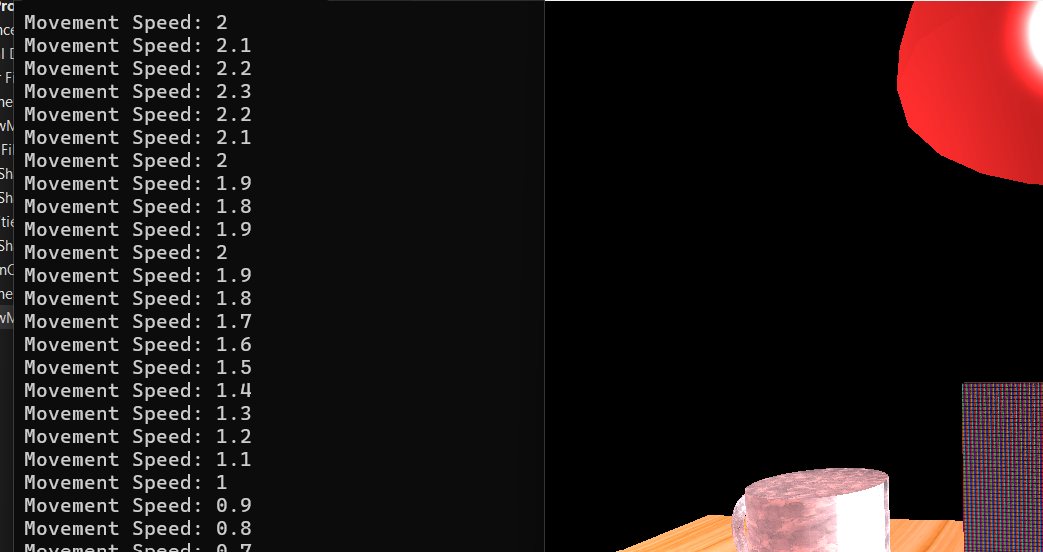
Figure 3The m\_MovementSpeed and gDeltaTime variables are used to determine how much movement should occur based on the frame time and the camera's movement speed.

Mouse movement is captured in Mouse\_Position\_Callback(), which calculates offsets based on the cursor's position and updates the camera’s direction dynamically. This provides a responsive, real-time navigation experience, making it easy for users to explore the scene from any angle.

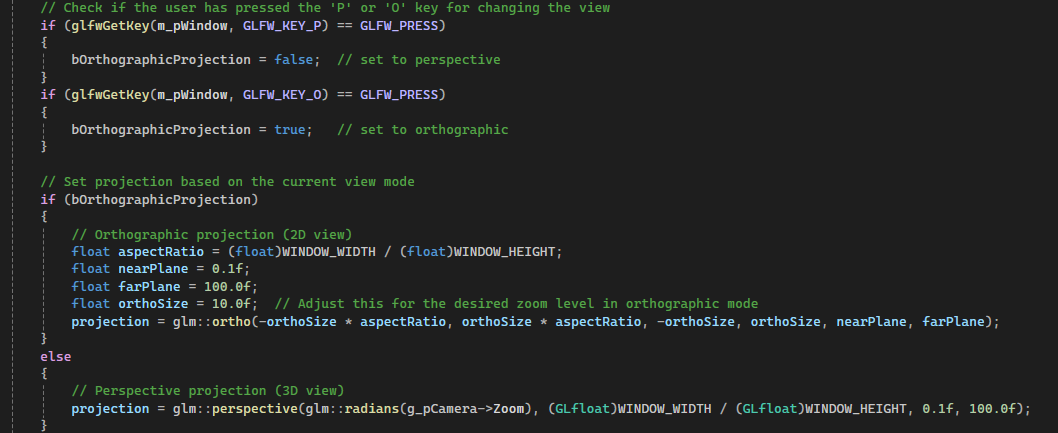


Mouse\_Position\_Callback() checks if it's the first mouse movement event (gFirstMouse). If it is, it records the current mouse position (xMousePos, yMousePos) into gLastX and gLastY, and sets gFirstMouse to false. This ensures that on subsequent calls, the current mouse position can be used to calculate movement offsets. After the first event, the method calculates the difference (offset) between the current mouse position and the last recorded position.

The Mouse\_Scroll\_Callback() function enhances usability by letting users dynamically adjust the camera’s movement speed via the scroll wheel. When the wheel is scrolled upward, the speed increases by 0.1, while downward scrolling decreases it by the same increment. To ensure smooth navigation, the function clamps the speed within a range of 0.1 to 10.0, preventing extreme values. This is complemented by console feedback, which displays the updated speed value, such as Movement Speed: 2.5, helping users monitor and fine-tune their navigation settings as each user will presumably have a different set-up and prefer differing native mouse sensitivity settings.



Another key feature is the ability to switch between 3D and 2D (orthographic) views, offering both depth perception and precise flat-plane visualization. This is implemented in the PrepareSceneView() function, where the P key sets the camera to perspective mode, and the O key switches to orthographic mode. These toggles dynamically adjust the projection matrix, catering to user needs, whether inspecting complex object relationships or to align elements accurately.



**Custom functions:**

There are four custom functions in my program, each creating a complex object (a cup, a laptop, lampshade, and mouse + mouse mat. These functions encapsulate the logic required to create and render specific objects, making the code modular, organised and easier to manage, reuse, and extend.

To take just one example, the void SceneManager::DrawCup(float posX, float posY, float posZ) function is designed to create and render a composite object—a cup—composed of a cylinder and a half-torus. By accepting position parameters (posX, posY, posZ), the function allows for flexible placement of the cup within the scene. Internally, the function sets transformations for scaling, rotation, and translation using local variables. It also contains all the texture and material values required to create the object. The parameters passed to the function determine the placement of the both the meshes that create the object such they are always relative to eachother like so:





This function, and the others like it, exemplifies modular design because it isolates all logic related to the creation and rendering of a cup object. Rather than duplicating code for transformations, texture settings, and rendering elsewhere, the function can be reused across the scene. The same function can be called with different parameters to place additional cups elsewhere. By centralizing the logic, such functions allow for transformations and rendering logic to be reused across multiple instances of the same object with different parameters, eliminating the need for repetitive code. Additionally, this modular approach simplifies maintenance, as any adjustments to materials or textures can be made in the function itself, rather than needing to be applied across multiple locations in the code.

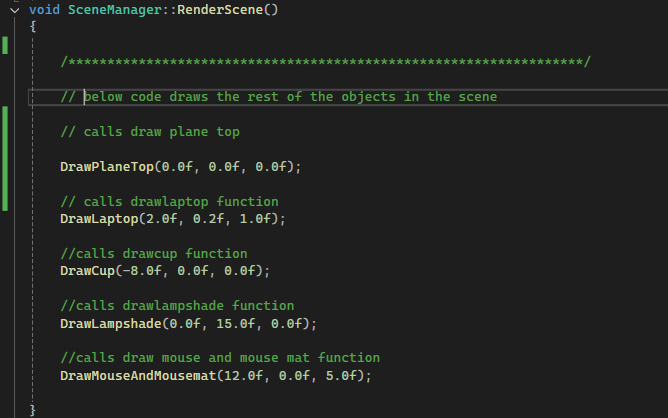


Figure 4 shows the function in which all objects are called to create the scene

The function could be further improved by adding an additional parameter to control the size (scale) of the object. This would allow for dynamic resizing of objects while maintaining their proportions. However, careful attention would be required to ensure that the scaling is applied accurately to both meshes that compose the object (such as the cup's cylinder and half-torus). Ensuring consistency in scaling between the meshes would be critical to avoid any misalignment or distortion, preserving the proportionality of the objects in the scene.