**CS 330 Project:**

*3D Scene with Camera Navigation and 2D/3D Toggle*

Charlotte Mancini

Southern New Hampshire University

CS 330 Computer Graphics & Visualization

Professor Jeff Phillips

December 10, 2023



**Design Decisions:**

When designing this 3D scene, which is written in the C++ programming language and employs Modern OpenGL, I began by choosing primitive shapes that closely resemble each real-world object. A plane was chosen to represent the desktop because both are somewhat flat rectangles, and after applying a woodgrain texture, the plane looks like a desktop. An elongated cube was selected to represent the computer monitor from the photo. It was built by creating a perfect cube, then scaling it up on the x axis and scaling it down on the z axis. With the image of an actual screen applied as the texture the object can now be perceived as a computer monitor.

For the complex object, I created a coffee mug consisting of two primitive shapes, first the cylinder, and then, a half-torus. The half-torus shape correctly represents the curved mug handle, while the body of the mug was best served by a short cylinder. The half-torus was the most complex shape although both shapes require distinct, custom functions to first ascertain the vertex and index datasets prior to rendering the individual shapes with the customized renderShape function, which is a milestone in the process that renders the entire scene.

These shapes (cylinder and half-torus) are then transformed to their appropriate places in the scene, and, as a complex object, the handle and mug must be translated to adjacent positions to create one complex object. To meet the requirements, the texture was created using an advanced texturing technique. Specifically, a blend of textures from more than one source was implemented.

When creating the Dunkin cup, I utilized one simple shape, a tapered cylinder, to best represent its tapered shape as seen in the real world. This also required a customized function using a complex looping algorithm to correctly calculate the vertices and indices prior to rendering the object with two other distinct custom functions (renderShape and renderScene).

For the ruler and pencil inside the mug, I utilized a cube and cylinder, respectively. The flattened cube represents the ruler, and the cylinder is placed in the coffee mug as the pencil to align with the original image.

**Navigation:**

To provide an enjoyable user experience, horizontal, vertical, and depth navigation are available. This navigation is controlled by direct user input via the keyboard and/or mouse. The camera can travel the X, Y, and Z axes using a set of nuanced controls to personalize the user’s view, and the camera’s orientation changes although its location remains unchanged. The user can move the mouse cursor to adjust this orientation so that they can look up, down, right, or left, and the mouse scroll can be used to adjust the speed of the camera as it travels around the scene.

Additionally, using the W, A, S, and D keys, the user can move the camera forward, backward, left, and right, respectively. Using the Q and E keys, we can control the upward and downward movement of the camera, and with the tap of the P key on the keyboard the scene toggles between orthographic (2D) and perspective (3D) views.

**Custom functions:**

To create the code base that enables this scene to display on the screen required much modularity because there are functions that must be repeated numerous times. Reusability is the essence of the DRY (Don t Repeat Yourself) principle adopted by most programmers and noted in coding best practices (Tarek, 2022). This practice is also especially helpful when debugging, and provides greater efficiency, eliminating endless lines of repetitious code (Pandey, 2022).

For this scene, multiple objects are rendered in the scene, but each one is created individually, with its own set of parameters, and yet they all use the same custom functions for rendering; this is an example of reusability. This technique promotes maintainability, readability, and improved security. The custom renderScene function accepts arguments that indicate the shape, size, texture, camera, view, and position, and it produces that requested object in its exact place in the scene. It contains a nested custom function (renderShape) to create each individual shape for the renderScene function to then transform and render accordingly. However, for the renderShape function to work properly, additional custom functions like the createHalfTorus and createTaperedCylinder functions were required to calculate specific vertices and indices data.

By nesting the renderShape function inside the renderScene function, we also adhere to the object-oriented programming principle of encapsulation, a secure coding best practice (GeeksforGeeks, 2023).

For your review, the function signature follows:

**void renderScene(GLFWwindow\* window, GLMesh& mesh, GLuint& shaderProgram, Camera& camera, bool isPerspectiveView, Texture& texture1, Texture& texture2, float blendFactor, TransformData& transformData);**

Another custom function in this program, applyProjectionAndMatrix, accepts arguments regarding the shader program, model Matrix, view Matrix, and whether the object should have an orthographic or projection view. This establishes if the Boolean argument, isPerspectiveView, is true or false, which then determines if this object will be rendered in 2D or 3D.

For greater modularity and efficiency of the source code, I also created a custom function to initialize and instantiate various fundamental components. The initApp function accepts the arguments that allow the program to set mouse callback functions, instantiate a virtual camera, and more. Rather than clutter the main, I combined these functions and replaced them with one line of code. As a preventative measure, and for exception handling, I added some custom functions that check and report any errors in detail. Custom shader programs were also added to integrate texture, lighting, and lighting color as required by the scene, and I created a SetLightProperties function that adjusts the parameters of light objects.

For the keyboard to control camera navigation, I implemented a custom function, processInput, that adjusts the camera view when the user selects a preset key on the keyboard (WASDQE), and if the user presses the P key, the view is transformed from 3D to 2D. By calling this custom function just once in the main function, every tap of the keyboard on any one of the personalized keys changes the scene instantly. This modularity provides an organized coding structure that aims to avoid an illegible labyrinth of functions in monolithic source code.

**References:**

Pandey, S. (2022, October 3). *Importance of code reusability in software development | BrowserStack*. BrowserStack. <https://www.browserstack.com/guide/importance-of-code-reusability>

Tarek, T. (2022, January 7). *Understanding the DRY (**Don’t Repeat Yourself) Principle*. Plutora. <https://www.plutora.com/blog/understanding-the-dry-dont-repeat-yourself-principle#:~:text=It%20basically%20states%20that%20two,refactoring%20and%20abstractions%20every%20time>.

GeeksforGeeks. (2023, September 4). *Encapsulation in C*. <https://www.geeksforgeeks.org/encapsulation-in-cpp/>