Scott Smith

Professor Rodriguez

CS-330 Computational Graphics and Visualization

8/3/2021

A picture containing text

Description automatically generatedA picture containing text, case, accessory

Description automatically generated

As a C++ and OpenGL 3D graphics developer working for Triangle & Cube Studios, I was recently tasked with creating a 3D render of a computer mouse setup for a client. The client requested that the 3D environment feature a trendy lighting aesthetic and include modern technology that would appeal to today’s PC gamers. Thankfully, I’ve been an avid gamer for most of my life so gathering the objects necessary for creating the mockup was as easy as looking down at my computer desk. The objects I’ve selected are just a few gamer must-haves, including a comfort-cushioned mousepad, mouse bungee, and high-speed gaming mouse, resting atop a clear-finished oak desktop. Seeing as this would be my first graphics development project for the company, the items I’ve selected were chosen to simplify the implementation enough to where I could generate the entire environment using only a handful of primitive shapes. See **Figures 1.1-1.2**.

**Figure 1.1 Figure 1.2**

A picture containing text, accessory, case

Description automatically generated

These images show my initial thought processes toward designing the scene. Nearly all these designs made it into the application, with minor exceptions. My original thoughts for designing the mouse bungee involved using a pyramid as a basis, but during development it became much easier to manually code the shape as its own “double-peaked” pyramid. See **Figure 1.3**.

**Figure 1.3**

Graphical user interface, text

Description automatically generated

The vertex array is like a primitive pyramid with a few additional vertices.

Another challenge was the implementation of the mouse. What I found most helpful for making a believable 3D rendition of its round-contoured shape was manipulating the model matrix, or more specifically, the *glm::scale* function. Manipulating the X, Y, and Z parameters allowed me to stretch and accent the curvature to match the look of a computer mouse. See **Figure 1.4**.

**Figure 1.4**

Text

Description automatically generated

With most of the heavy-handed mathematics being applied through external APIs, I was able to focus deeply on the aesthetics of the 3D environment.

Since the client required the scene to have a direct appeal to modern gamers, traditional WASD keyboard control as well as additional movement and perspective controls using mouse input were added to the render. See **Figures 2.1-2.2**.

**Figure 2.1**

Text

Description automatically generated

**Figure 2.2**

Text

Description automatically generated

Movement controls using the WASD keys work by calculating the camera’s current orientation with a pre-defined *velocity* float value consisting of a set movement speed and the time that has passed since the last frame. The movements, which correspond with the WASD keys in the traditional forward, left, backward, and right directions simply adjust the camera by the product of *velocity*. Additionally, movement control to direct the camera up and down using the Q and E keys has also been incorporated to establish greater control while navigating the scene and the mouse cursor can be used to adjust the orientation of the camera. If this wasn’t enough, users can speed or crawl around the 3D world using the mouse scroll-wheel to set the movement speed manually. See **Figure 2.3**.

**Figure 2.3**

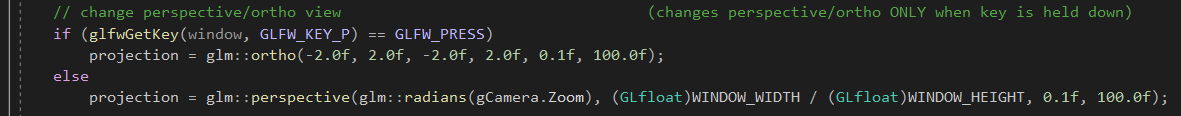
**Text

Description automatically generated**

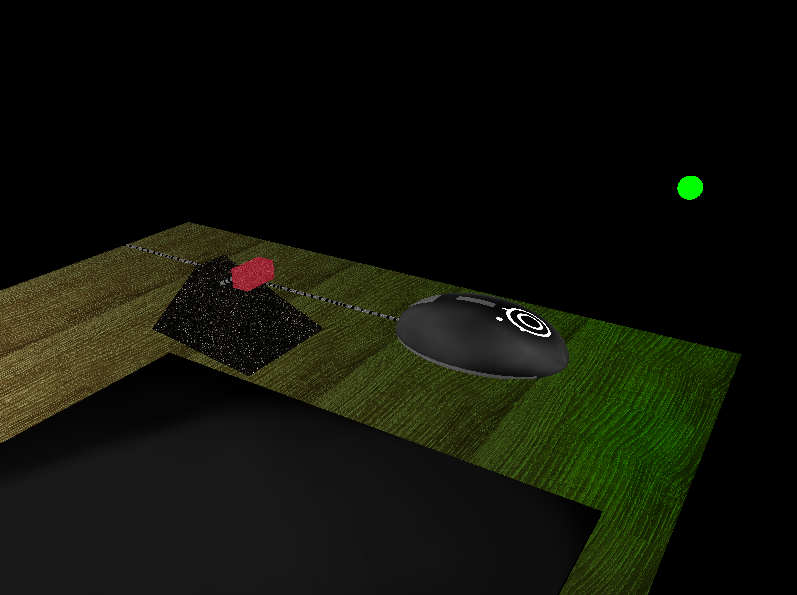
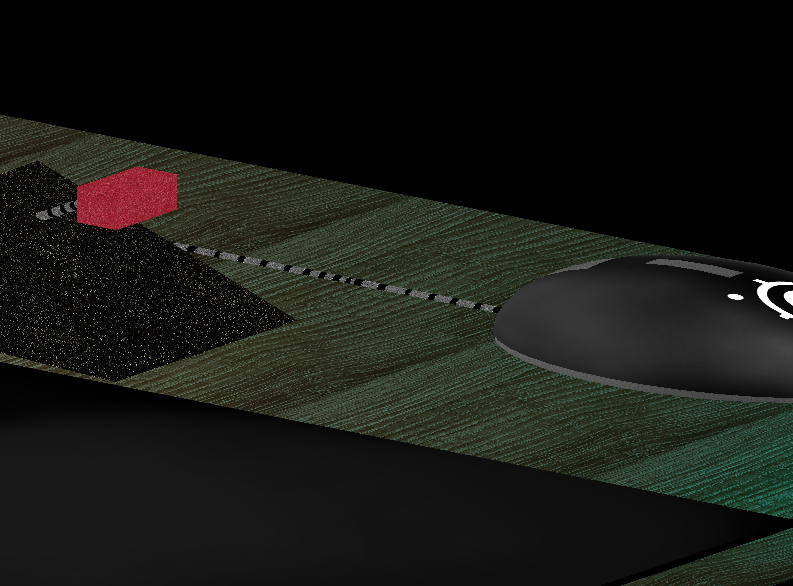
Movement speed is controlled to stop movement entirely or limit erratic behavior.

Another unique feature is the ability to change the camera’s perspective from a 3D environment to a 2D environment. As a user holds down the P key, they will be able to navigate the environment in 2D. Releasing the P key will return the user to the 3D environment. See **Figures 2.4-2.6**.

**Figure 2.4**



**Figure 2.5 Figure 2.6**

**Figure 2.5** shows the environment in 3D, **Figure 2.6** shows the environment in 2D.

To make code as reusable as possible, all object meshes are separate *GLMesh* structs, and each mesh is implemented using its own generator function called in *main()*. See **Figures 3.1-3.3**.

**Figure 3.1 Figure 3.2**

**Text, chat or text message

Description automatically generated** **Text

Description automatically generated**

**Figure 3.3**Text

Description automatically generated

Using a modular design polishes the implementation and makes for a more readable and easier-to-navigate code base. I entertained the idea of designing a single mesh generation function that would receive different vertex arrays to create each of the unique 3D shapes in the environment, but since I was using external APIs for some of the complex shapes, such as the cylinders and spheres, I decided this universal function might be counterproductive and could potentially lead to nested logic errors later. It’s simple enough to navigate to each specific object and investigate how it is being generated, whether it uses a set of vertices, indices, or learn how its buffers and vertex attributes are being assigned.

The heart of this implementation lies in the *URender()* function. In this function, I connect all meshes to their appropriate shaders, transform and position them using specific model, view, and projection matrices, bind each vertex array and corresponding texture, then draw the meshes in the window using primitive types such as *GL\_TRIANGLES*. The beauty of using a modular design pays off at this stage of development. Creating additional objects in the scene simply involves generating a unique model matrix and binding the previously generated mesh vertex array. In retrospect, features of this function could have been encapsulated a bit further, as each mesh follows a nearly identical pipeline during execution. Again however, I found it more intuitive to keep each render within the same function as specific shader attributes need be applied at different times. Unique examples involve applying multiple textures using a *bool* value for the back face of the mouse bungee and tiling a basic texture to fill out a larger object. See **Figures 3.4-3.8**.

**Figure 3.4**

**Text

Description automatically generated**

Apply multiple textures.

**Figure 3.5**

Text

Description automatically generated

Tile (scale) a basic texture.

**Figure 3.6**

Text

Description automatically generated

Fragment shader logic.

**Figure 3.7**

**A picture containing text

Description automatically generated**

Multiple textures can be seen on the back face of the mouse bungee. The mouse cable also uses the same texture used on the smaller cylinders of the mouse bungee. Using a scaling feature passed to the fragment shader source creates a realistic looking mouse cable.

To create multiple light sources more easily in my 3D scene, I decided to encapsulate the Phong lighting model into its own function, *generatePhong()*. This function returns a vector result of the Phong lighting model and can be passed arguments that specify a light source’s color and position. There are currently two lights that are being reflected by the clear-finished computer desk texture, a bright white light and a dimmer RGB style (color-oscillating) light source. Though this function is present within the fragment shader developed for the desk’s light reflections, it can easily be plugged into other applications that utilize GLSL to facilitate the generation of light upon different objects or textures. See **Figures 3.8-3.10**.

**Figure 3.8**

**Text

Description automatically generated**

The *generatePhong()* function calculates a customizable Phong result vector.

**Figure 3.9**

**Text

Description automatically generated**

*generatePhong()* is called by the main function of the desk object’s fragment shader and creates two unique light sources.

**Figure 3.10**

**Text

Description automatically generated**

A simple function that uses *glm::rotate* to oscillate between different colors based on existing light source animation code. *gLightColor2* is passed to both the desk’s fragment shader as well as the RGB light source’s fragment shader and allows both shaders to match each other precisely in the 3D environment.