CS 330 Design Decisions

Faris Malik

SNHU CS-330

Dec 11th, 2024

The development of my 3D scene was influenced by the need to balance functionality and aesthetics while meeting project requirements. To get me started I created the primary objects in the scene with simple geometric shapes like cubes, spheres, and cylinders. Why these shapes, because they are easy to work with and space efficient to render.

I created the lamp using a mix of cylinders and a cone- having a point light at the location of the lamp to help give the same effect as the reference photo did. I did also use directional as well as ambient lighting to make the scene look visually appealing. It helped to give depth to this lighting setup. I customized the light to look as sophisticated as possible with different intensities and colors to give the scene that amber hue- just like the reference. As for the textures, I used the tiling method on the wall to give it that nice look without pixelation, regardless of how big the wall is, choosing a seamless texture for it so that it has no visible seems where each texture tile ends.

To view the scene, I implemented both orthographic and perspective projections. Orthographic projection helps with precision tasks by showing objects without distortion, while perspective projection creates a more realistic and immersive view. By pressing the O and P keys users can switch between these modes. This freedom means that the scene can be used for general exploration and also for technical tasks.

Navigation within the scene was designed to be intuitive. The WASD keys let you move forward, backward, left and right, Q and E keys for moving up and down. To look around the scene, the mouse is used and scroll wheel for zooming in or zooming out. They are wonderfully combined controls to explore the environment and to inspect details.

**Modular Design and Custom Functions**

To keep the code organized and easy to maintain, I created multiple custom functions. These functions do a particular job, and it cleans up the main program and enables addition of new features later.

One important function is ProcessKeyboardEvents(), which processes all of the keyboard input about moving the camera and toggling projection modes. All keyboard related actions are taken care of with this single function which makes it easier to update or enhance.

Another key function is Mouse\_Position\_Callback(), which captures mouse movement to control the camera’s orientation is another key function. The function only runs in perspective mode in which looking around the scene is needed. By isolating this behaviour it'd be easy to adjust the code for that other devices like a VR headset.

I defined Scroll\_Callback() for zooming, changing the camera’s zoom level depending on input from the scroll wheel. This functionality lets us concentrate on a particular object or get the bigger picture.

I could have encapsulated the creation of the lamp so that it would be easier to have multiple created incase of a complex scene with multiple lamps, but luckily, we needed only one for this scene. Though, that would still be helpful if needed in another program.

Finally the setup of the camera and projection matrices is handled in PrepareSceneView(). Its purpose is to make switching between orthographic and perspective views seamlessly and reliably. With encapsulating these tasks we are able to reuse this function in other projects or change it and it wont affect the other parts of the code.

**Implementation Challenges and Results**

The biggest challenge in the project was making sure that all the controls would work together well. One example: Scroll\_Callback() had to work correctly in the both projection modes to have a nice and smooth zooming experience. Ensuring smooth interaction required careful attention to detail in coordinating input events with the rendering loop.

The ProcessKeyboardEvents() function processes captured keyboard inputs to enable the user to control the camera movement and position as seen below:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* ProcessKeyboardEvents()

\*

\* This method captures and processes different inputs.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void ViewManager::ProcessKeyboardEvents() {

if (glfwGetKey(m\_pWindow, GLFW\_KEY\_ESCAPE) == GLFW\_PRESS) {

glfwSetWindowShouldClose(m\_pWindow, true);

}

if (!g\_pCamera) return;

// Process camera movement

if (glfwGetKey(m\_pWindow, GLFW\_KEY\_W) == GLFW\_PRESS) g\_pCamera->ProcessKeyboard(FORWARD, gDeltaTime);

if (glfwGetKey(m\_pWindow, GLFW\_KEY\_S) == GLFW\_PRESS) g\_pCamera->ProcessKeyboard(BACKWARD, gDeltaTime);

if (glfwGetKey(m\_pWindow, GLFW\_KEY\_A) == GLFW\_PRESS) g\_pCamera->ProcessKeyboard(LEFT, gDeltaTime);

if (glfwGetKey(m\_pWindow, GLFW\_KEY\_D) == GLFW\_PRESS) g\_pCamera->ProcessKeyboard(RIGHT, gDeltaTime);

if (glfwGetKey(m\_pWindow, GLFW\_KEY\_Q) == GLFW\_PRESS) g\_pCamera->ProcessKeyboard(UP, gDeltaTime);

if (glfwGetKey(m\_pWindow, GLFW\_KEY\_E) == GLFW\_PRESS) g\_pCamera->ProcessKeyboard(DOWN, gDeltaTime);

The controls of the camera and projection mode needed to integrate carefully. The WASD and the QE keys provide smooth and responsive movement whereas the mouse controls are for a neat orientation adjustment. WASD controls the following directions of the camera: W makes you go forward, A is for going left, S is to go backwards, and D is to go right. And Q & E control the height of the camera, with Q being up and E being down. The view matrix gets switched between projection modes dynamically, but switching also updates the camera’s position and orientation.

Overall, the project resulted in a 3D scene that is both functional and visually appealing. The modular design and user-friendly controls make it easy to navigate and interact with, providing a solid foundation for future improvements or additions.