The development choices for my 3D scene were guided by both visual aesthetics and functional goals. I selected a combination of basic geometric shapes—cubes, planes, and cylinders—to construct a stylized outdoor environment, including architectural structures and natural elements. These shapes were chosen for their simplicity, rendering efficiency, and the ease with which they could be textured. Textures such as wood, stucco, glass, and grass were applied to enhance realism while maintaining optimal performance for real-time rendering. The original image I proposed had absolutely no color to it and so the process of what I would imagine my design as took a lot of decisions.

Functionally, I prioritized modularity and interactivity. The `SceneManager` class encapsulates all logic related to scene setup and rendering, including loading shaders, binding textures, and drawing models. This modular approach allows me to manage assets and rendering logic in a centralized way, making the codebase easier to expand or debug.

To navigate the 3D scene, users can use the keyboard and mouse. The `ViewManager` handles the virtual camera, providing free-look movement with the WASD keys and mouse input for rotation. Mouse movement captures changes in x and y position to adjust yaw and pitch, simulating a first-person perspective. Additionally, scroll wheel input adjusts the field of view for zooming. This functionality was programmed using GLFW’s input callback system, ensuring responsive, real-time controls. Including added controls for both Q and R keys to move up and down as needed.

The virtual camera implementation uses a custom `Camera` class that supports view matrix generation based on user input. This camera system is initialized in `ViewManager`, which also updates camera movement using delta time to maintain smooth transitions regardless of frame rate. The camera direction vector is updated on mouse movement, creating an immersive navigation experience that feels intuitive for users familiar with 3D environments.

Modularization is achieved through several custom functions, such as `setupScene()`, `renderScene()`, and `processInput()`. These functions abstract specific responsibilities: scene creation, object rendering, and input processing, respectively. This separation of concerns reduces redundancy and allows for code reuse. For example, `processInput()` is designed to be reusable across frames, cleanly interpreting keyboard input to manipulate the camera’s position.

In summary, my development approach emphasized clarity, interactivity, and extensibility. The code was organized around distinct modules for scene setup, camera handling, and rendering. User interaction was designed with accessibility in mind, enabling intuitive navigation through mouse and keyboard. The result is a cohesive, responsive, and visually engaging 3D scene.

References

LearnOpenGL. (n.d.). \_Camera\_. https://learnopengl.com/Getting-started/Camera

OpenGL. (n.d.). \_The Industry Standard for High Performance Graphics\_. https://www.opengl.org/