Damon Harris

06/22/2025

Reflection

**Justify Development Choices: The 3D Scene**

For my 3D scene, I modeled a Star Trek mug placed on a circuit-textured floor, complemented by decorative elements such as gold rings and a uniquely designed handle. These objects were selected because they allow me to demonstrate a variety of 3D mesh types of cylinders, tori, pyramids, and spheres while applying transformations (scaling, rotation, and translation) in a meaningful and visually balanced manner.

The handle design, composed of two pyramids connected by a glassy sphere, provided an opportunity to experiment with custom object positioning and lighting effects on both metallic and reflective surfaces. I implemented this functionality using reusable transformation functions, custom material definitions, and texture mapping, all managed through a modular SceneManager class.

The planet background image is using the m\_basicMeshes->LoadBoxMesh(); and in the RenderScene() I doubled the scale using: glm::vec3 scalexyz = glm:: vec3(100.0f, 50.0f, 10.0f); this ensures that it is 100 units wide, 50 units tall, and 10 units thick while it sits behind the scene so it do not overlap other objects.

**Explain How a User Can Navigate Your 3D Scene**

Camera control in the 3D scene is implemented via a virtual camera defined in the ViewManager or a related input-handling module (not shown directly in SceneManager.cpp . The user’s viewpoint is determined by setting the viewPosition vector in the shader, which simulates the eye or camera location.

For input, the setup is compatible with both mouse and keyboard controls using GLFW callbacks. This allows users to pan, zoom, and rotate the view interactively. The interaction model mirrors that of real-time 3D engines, providing an immersive and intuitive scene navigation experience.

**Explain the Custom Functions That Make Your Code Modular and Organized**

Several custom functions in SceneManager.cpp enhance modularity and code organization:

* SetTransformations(...): Applies scaling, rotation, and translation to any mesh before rendering. This function is reusable for all objects and eliminates redundant transformation code.
* SetShaderMaterial(...) and SetShaderTexture(...): Abstract the logic for binding material properties and textures, enabling consistent and reusable shader setup across objects.
* CreateGLTexture(...): This function manages the loading, configuration, and assignment of textures, encapsulating complex OpenGL texture logic into a single, reusable function.
* PrepareScene() and RenderScene(): These functions separate scene setup (loading meshes and textures) from the actual rendering logic, keeping the codebase clean and easy to navigate.

By separating rendering logic from object definitions and following the DRY (Don't Repeat Yourself) principle, these custom functions improve maintainability and make it easy to expand the scene. For example, adding a new object only requires defining its transformations and calling a draw method.

**References**

Chua, E. H. (n.d.). Computer graphics: Basic theory. Nanyang Technological University. Retrieved from <https://www3.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html>  
Joey de Vries. (n.d.). Transformations. Learn OpenGL. Retrieved from <https://learnopengl.com/Getting-started/Transformations>  
GLFW. (n.d.). Input guide. Retrieved from <https://www.glfw.org/docs/latest/input_guide.html>  
docs.gl. (n.d.). OpenGL API documentation. Retrieved from <https://docs.gl/>