# CS 330 Final Project Reflection – OpenGL

# Daniel Gorelkin - Aug 9, 2025

**Image Selection and Development Choices**

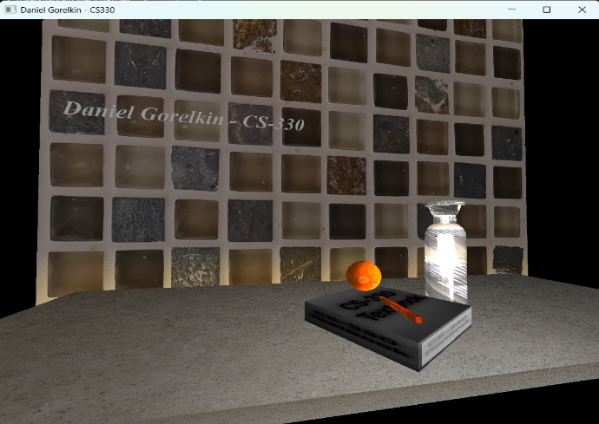
The image I will use in my final project to create a 3D-rendered scene will present four stacked items placed on a flat table in front of a wall-like background, as shown in the following sketch image.

 A lemon on a notebook

AI-generated content may be incorrect. A lemon on a notebook and a pencil on a tile floor

AI-generated content may be incorrect.

My scene will contain all the items from the sketch, e.g., the base plane, the backdrop wall, the pitcher, the book, and the pencil, on top of and beside one another. These items would be a good choice for my project, as they can be constructed from basic 3D shapes, such as a box, cone, cylinder, sphere, and torus. This scene will mimic a good practice of displacing objects in a stacked position, where some objects obscure others and provide a realistic 3D visualization. For example, rendering the base and the backdrop objects can be done from a basic boxed shape placed perpendicular to one another. A more complex object, like the pitcher, will be constructed from a series of basic objects, such as a torus, cylinder, and half a sphere stacked one on top of the other and placed over the boxed base plane. That functionality will be achieved by using OpenGL libraries such as glad, glew, glfw, and glm. To begin with, I will initialize my scene window by calling InitializeGLFW() from the glfw library, and instantiate the ShaderManager() that will be responsible for compiling and linking GLSL shader programs from vertex, fragment, geometry, etc., shader source files. Next, I will initialize my ViewManager() and pass it the shader object, which will act as a controller that tells the engine what to draw and how to draw it and control the camera in my scene. Lastly, by calling the SceneManager() that will be responsible for arranging the objects in the scene, their appearance, and the lighting, I will call the RenderScene() method that will bind all the above together and create a functional scene window, which will output the following functionality:



**Navigation and Rendering**

To navigate the 3D scene, I will set up my ViewManager() class. It will be responsible for initiating the camera to view the objects in the scene and navigating it. The Camera() class inside the ViewManager() defines the camera's position and direction, as well as its perspective (e.g., orthographic or perspective views). The main difference between the views lies in how they project 3D objects onto a 2D screen and how they handle the depth and size of the scene through the view matrix. The ViewManager class will also be responsible for the input controls that bring the scene to life. To control my virtual camera, I will use keyboard and mouse inputs. The WASD keys will control moving the camera up, left, down, and right, respectively, while the Q and E keys will shift the camera up and down on the Y-axis. In fact, my virtual camera will be located in a fixed location, and the objects in the scene will be placed in a new position compared to the camera location. That functionality will be achieved by the ProcessKeyboardEvents() method, which will listen to input events and send requests to the ShaderManager() class. Similarly, the O and P keys will switch between the orthographic and perspective projection views, which will change the depth of the scene and provide a sense of distance on the 2D output display screen. Additionally, the scene will be responsive to the mouse control inputs via the Mouse\_Position\_Callback() method. The mouse will update the direction at which the camera is looking, and by scrolling the mouse wheel or holding the Ctrl key button and scrolling the wheel, will change the speed at which the camera “moves” in the scene and how sensitive it is to the mouse movements.

A screen shot of a computer

AI-generated content may be incorrect.

**Functionality and Reusability**

To make my code functional and reusable in various implementations, my code follows the object-oriented principle to allow easy modification and upgrade of any of the above-mentioned components. For example, adding more meshes or modifying existing ones can be easily done through the ShapeMeshes.cpp file, which holds the definition of all the basic objects for the scene. Hence, any more complex object, such as a 3D building image, can be defined by a set of millions of triangles and vertices and represented as a wireframe.

A screenshot of a computer screen

AI-generated content may be incorrect.

Going further, each of the loaded mashes can be sculpted by wrapping it in a custom-defined texture and applying lighting to it through shaders and material setup from the standalone SceneManager.cpp file that contains the SetShaderColor(), SetShaderTexture(), SetTextureUVScale(), SetShaderMaterial(), and SetupSceneLights() methods, which could be abstracted or defined individually for reusability purposes.

A scale with a ball and a cone

AI-generated content may be incorrect.

Similarly, the ViewManager() class, which resides in its own file, can be easily replaced to enable scene navigation using a joystick or a virtual reality device. The same modularity principle applies to upgrading and modifying OpenGL library files, enabling expanded functionality and improved performance in more complex scene rendering environments.

**References**

*GLFW: Input guide. (n.d.).* [*https://www.glfw.org/docs/latest/input\_guide.html*](https://www.glfw.org/docs/latest/input_guide.html)

*LearnOpenGL - Introduction. (n.d.).* [*https://learnopengl.com/Introduction*](https://learnopengl.com/Introduction)

*Seal, R. S. (2025, February 8). Still Life drawing class. Pinterest.* [*https://www.pinterest.com/pin/19914423347254135/*](https://www.pinterest.com/pin/19914423347254135/)