Tanner Gaudes

CS-330

02/13/2023

**Final Project Reflection**

**Justify development choices for your 3D scene.**

The photo in which the virtual scene is based was deliberately populated with objects that are simple to recreate in OpenGL. Many primitive objects like the plane, cube, and pyramid are easy to recreate in code because the number and placement of vertices does not change regardless of the object’s size; this made the creation of the floor planes, the battery body, the deck of cards, and the brick pyramid easy to create. However, generating cylinder vertices is a much more complex task because the number of vertices in a given cylinder (or any object consisting of round faces) changes based on the desired resolution. The circles on the top and bottom of the cylinder must be generated dynamically with the help of trigonometry and then joined using the necessary triangles. Additionally, generating the normal vectors and texture coordinates for each vertex is not trivial.

Unfortunately, generating a torus object programmatically proved to be a problem that I was unable to solve in any reasonable amount of time. Thus, the rubber band (a torus) is instead represented as a cylinder for the sake of development complexity. To meet the requirements outlined in the project rubric (specifically, the requirement to use at least four unique shapes), an additional textured pyramid object was inserted on the lower-right of the plane. Note that this object does not exist in the original photo that the virtual scene is based on.

**Explain how a user can navigate your 3D scene.**

This OpenGL application implements free camera movement that allows the user to easily navigate around the virtual scene. Basic movement can be done using the W, A, S, and D keys on the keyboard: W and S move forward and backward, and A and D move left and right, respectively. Additionally, the E and Q keys allow the user to move the camera directly upward or downward on the Y-axis (relative to the camera’s facing direction). The camera’s facing direction is controlled by the mouse location in respect to the application window: Moving the mouse up or down will adjust the camera pitch up and down, and moving the mouse left or right will adjust the camera yaw left or right; the camera’s roll is locked. To help the user navigate the scene more easily and efficiently, the mouse wheel can be scrolled up or down to increase or decrease the speed of camera movement (but not rotation). This can be useful when the user needs to move a long distance or wants to get a very close-up view of an object. Finally, the P key on the keyboard can be pressed to toggle the camera’s projection mode between perspective and orthographic projection.

**Explain the custom functions in your program that you are using to make your code more modular and organized.**

Naturally, the application’s code is split into a series of separate functions to enhance readability and modularity. Most notably, as numerous different mesh objects need to be created at runtime, a separate function to generate every individual mesh was created that requires only a single parameter-less call to generate each of the application’s necessary mesh objects and store them in variables. Similarly, a function to generate each individual texture object was created with the same concepts and methods implemented. Within both functions is a series of nested function calls to a function that generates a *single* mesh and assigns the data to its corresponding variable; handling these as nested function calls helps prevent visual clutter within the main function. A similar concept is used in the function to delete each previously generated mesh and texture upon application exit.

Each type of mesh requires a unique set of vertices to display correctly. Thus, separate functions were created for each type of mesh used: Plane, cube, pyramid, and cylinder. Within these functions, all necessary vertex data (relative position, normal vector, and texture coordinates) is created and passed into another function responsible for the instantiation of and interaction with this mesh data using built-in OpenGL functions. Additionally, a fully modular function was created specifically to generate a model matrix for any given object using the specified scale, rotation, and translation on each individual axis. This has proven to be extremely useful because it eliminates a lot of unnecessary and hard-to-read visual clutter from the main render function.

Last but certainly not least, the main render function was dramatically altered to make use of other modular functions like the one used to automatically generate model matrices. That said, there is significant room for improvement within the render function because it remains highly visually cluttered. For example, the multiple steps required to activate and bind vertex array objects could be organized into a single parameterized function to dramatically reduce code repetition. Also, rendering using different shader programs could be split into separate functions to enhance code readability. Furthermore, data corresponding to meshes and textures like position and the objects themselves could be stored within custom Objects and stored in iterable containers to allow looping over the contents directly, enhancing code readability and expandability even further.