Craig O’Loughlin 10/16/2022 CS330

**7-1 Project: Reflection**

In this project we created a 3D scene by coding directly into an OpenGL API using C++. There were many interesting challenges involved in this and it was a great opportunity to learn about how computer graphics work at a low level.

Development Choices

My scene consisted of a few basic objects of which I selected in order to include a variety of primitive shapes. There are two stacked card decks, a rubber ball, and a candle jar. There is also a toy duck which I included in order to attempt building a more complex shape using primitive ones. The OpenGL API allows for a few different approaches to rendering shapes, but I chose to render everything as regular triangles by grouping sets of vertices into index points. This simplified the rendering function so that only one function was needed that can accept any shape. In untextured objects this approach also can save GPU memory, as vertices can be reused in the index array. However, all objects in my scene are textured, and indices cannot share vertices here because they may carry separate texture mappings. Finally, this approach allowed for uniform mesh generation between different shapes, to where I was able to use one base class for all shapes.

Textures were loaded as images using a pre-built open library, and this simplified the process of converting images into byte arrays. Lighting was handled using the Phong lighting method, a mixture of ambient, diffuse, and specular light that is calculated per pixel in the fragment shader. This method is simple and effective for basic graphics. There are two shapeless point-lights in my scene that work to properly illuminate objects and provide specular details. I chose to implement specular intensity directly into the material object, so that each material can define its own level of specular intensity to the render function. This allows a visual difference between glossy and matte surfaces. Another approach would be to include a specular map for each object, which can inform the shader of specular intensity per pixel; and this would be more accurate but was for the most part unnecessary for my scene.

Finally, I am happy to say that I was able to work out cylinder mesh generation programmatically without any resources. This was a fun challenge. The result is cylinders with any number of edges as defined at creation (although the resulting meshes are quite large). For spheres, I did have to use a guide, but I learned I had the correct approach of a circle rotated n times and was happy to understand the concept enough to modify the resource’s algorithm to my needs.

Navigation

The camera view of the scene was created using a model – view – projection approach, where each is a matrix of positional vectors. The main camera is a point in space along with a set of vectors for position and rotation. The scene is rendered at the camera’s view by multiplying the model matrix by the camera’s vectors. This is done in the vertex shader.

Navigation of the scene by moving the camera is possible by using a combination of mouse and keyboard control. The GLFW library was used to capture the hardware movement as an event and then call the associated activity for that event. The mouse is a different input device than a keyboard, and processing its movement was a separate challenge from keyboard presses. While the keyboard buttons are monitored after each frame by direct polling, the mouse is setup as an event-handler system. Mouse direction is ported from the GLFW library into the handler function, and then these are translated into 3D camera rotational directions using trigonometry.

Another challenge regarding navigation was implementing both a perspective and orthographic view at the press of a button. Simply polling a key does not work here as the view will toggle rapidly between views as long as the associated key is pressed. Instead, a separate callback needed to be created that activates once and only on key down press. GLFW has the ability to distinguish this event by default.

Custom Functions

My goal for the architecture of this project was to build an abstraction layer on top of the OpenGL GLEW and GLFW libraries, so that the main function concerns itself only with choosing the shapes, positions, textures, and lighting to render. For this, I created an OpenGLController object that interfaces with GLEW. I used object-oriented principles as mush as possible in order to create encapsulation, although looking back this approach was likely a bit too rigid for the application and didn’t mesh well with some of the API, particularly callbacks. This approach also likely slows down the scene setup (which still ran quite quickly on my computer). To address this, I could instead pass more information by reference to avoid making expensive copies of large meshes.

The OpenGLController also defines a set of shapes, as mesh arrays with some accompanying information, as OpenGLMesh objects. This makes the shape definition process uniform. The main function can define a new object, assign a shape to it from the shape namespace, and then set parameters directly on the object. Finally, the OpenGLController accepts the mesh in the AddShape() method, at which point the data is loaded into GPU memory and readied for rendering. A single method, .RunScene() is called after all shapes are added, and loops until the window is closed or the esc key is pressed. The loop renders a frame, then polls inputs, then repeats.

Overall, I feel that the architecture is beneficial to scene creation and that the result is reusable by anyone who would want to draw objects on a scene. For example, I was able to make use of the controller myself when creating the toy duck in my scene; as the last object, it was simple to define some shapes from the main function and combine them by translation and rotation. I did not have to think about meshes and rendering as it was handled in the controller already.