Term Milestone Project: Creating a 3d Scene

Jared Semonin

CS-330 Computer Graphics and Visualization

Professor Phillip Enkema

June 24th, 2023

In my paper, I carefully selected specific objects for my 3D scene to showcase my proficiency in rendering intricate 3D shapes using the OpenGL framework. The tape measure was chosen as a prime example to demonstrate the utilization of a cylinder and a cube in constructing a complex 3D shape. Similarly, the Star Wars droid was specifically chosen to illustrate the implementation of two spheres, while the hard drive dock exemplifies the utilization of a straightforward elongated cube. Lastly, the headphones were deliberately selected to showcase the integration of spheres, cylinders, and toruses in their creation.

Creating a sphere in OpenGL I found to be the most challenging, and I personally found it to be the most difficult task. I'm not entirely sure whether it was due to my unfamiliarity with OpenGL and C++ or if the process itself is inherently complex. However this is the process that I came across while facing these challenges

1. Start by defining the necessary variables:
   * Radius: This determines the size of the sphere.
   * Slices: Controls the number of horizontal divisions you want on the sphere.
   * Stacks: Determines the number of vertical divisions you want on the sphere.
   * Theta: Represents the angle for the horizontal division.
   * Phi: Represents the angle for the vertical division.
2. Generate the vertices and indices for the sphere:
   * Use nested loops to iterate through the stacks and slices.
   * Calculate the phi angle using the formula phi = π \* i / stacks, where i represents the current iteration.
   * Within the inner loop, calculate the theta angle using the formula theta = 2 \* π \* j / slices, where j represents the current iteration.
   * Use spherical coordinates to calculate the x, y, and z coordinates of each vertex:
     + x = radius \* sin(phi) \* cos(theta)
     + y = radius \* cos(phi)
     + z = radius \* sin(phi) \* sin(theta)
   * Store the vertex coordinates in a data structure or array.
   * Calculate and store the indices that define the triangles for rendering the sphere.
3. Render the sphere:
   * Set up the necessary buffers (VAO, VBO, EBO) and configure the vertex attributes.
   * Use a rendering loop to draw the triangles defined by the indices.
   * Apply any desired transformations, such as translation or rotation, to position the sphere as needed.

This is the basic creation of the sphere this process may change with the addition of adding textures or lighting. I am still unfamiliar as to how those may effect this process.

To navigate a my 3D OpenGL scene using the keyboard and GLFW, these are the steps that I followed:

1. Initialize GLFW:
   * Set up a GLFW window and context.
   * Set the desired window properties, such as width, height, and title.
   * Initialize GLFW using **glfwInit()**.
   * Create a GLFW window using **glfwCreateWindow()**.
2. Set up keyboard input callbacks:
   * Set the keyboard input mode using **glfwSetInputMode(window, GLFW\_STICKY\_KEYS, GLFW\_TRUE)**.
   * Set up a keyboard callback function using **glfwSetKeyCallback(window, keyCallback)**.
3. Implement the keyboard callback function:
   * Define a function that will be called when a key is pressed or released.
   * Use this function to handle the keyboard input and perform the desired actions for navigation.
4. Handle navigation actions:
   * Keep track of the camera position, orientation, and movement parameters.
   * Update the camera based on the keyboard input received.
   * For example, you can use the following keys to control camera movement:
     + W: Move forward.
     + S: Move backward.
     + A: Move left.
     + D: Move right.
     + Q: To move up.
     + E: to move down.
5. Implement the main rendering loop:
   * Use a **while** loop to continuously render the scene and handle user input.
   * Check for keyboard input and update the camera accordingly.
   * Clear the buffers using **glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT)**.
   * Render my 3D objects using the appropriate transformations and shaders.
   * Swap the front and back buffers using **glfwSwapBuffers(window)**.
   * Poll for events using **glfwPollEvents()**.
6. Terminate GLFW:
   * When done with rendering, terminate GLFW using **glfwTerminate()**.

The same process is followed when using a mouse and cursor to move the camera

1. Set up mouse input callbacks:
   * Set the mouse input mode using **glfwSetInputMode(window, GLFW\_CURSOR, GLFW\_CURSOR\_DISABLED)**.
   * Set up a mouse callback function using **glfwSetCursorPosCallback(window, cursorPositionCallback)**.
   * Declare and initialize variables to store the previous mouse position.
2. Implement the mouse callback function:
   * Define a function that will be called when the mouse moves.
   * Calculate the mouse's movement delta since the last callback.
   * Update the camera's orientation based on the mouse movement delta.
   * Calculate the new camera's look-at vector or rotation angles.
3. Handle camera movement:
   * Keep track of the camera position, orientation, and movement parameters.
   * Update the camera's position and orientation based on keyboard input (similar to the previous keyboard navigation).
   * Use the updated camera parameters to generate the view matrix.

Throughout my project I don’t believe I used very many custom functions as I heavily relied on the CS330 github tutorial code as a template of sorts for my project. Some of the fundamental functions used throughout my projects are:

1. **glClear()**: This function is used to clear the buffers, such as the color buffer and depth buffer, before rendering a frame. It takes flags as parameters to specify which buffers to clear.
2. **glUseProgram()**: This function is used to activate a specific shader program for rendering. It takes the shader program object as a parameter.
3. **glBindVertexArray()**: This function binds a vertex array object (VAO) that stores the configuration of vertex attributes. It specifies the format of vertex data and how it should be interpreted.
4. **glBindBuffer()**: This function binds a buffer object that holds vertex data, such as vertex positions, normals, or texture coordinates.
5. **glEnableVertexAttribArray()**: This function enables a specific vertex attribute array for rendering. It takes the index of the vertex attribute as a parameter.
6. **glDrawArrays()**: This function draws primitives using vertex data stored in the currently bound VAO and vertex buffer objects (VBOs). It takes the primitive type, starting vertex, and the number of vertices as parameters.
7. **glDrawElements()**: This function is similar to **glDrawArrays()**, but it uses an index buffer object (IBO) to specify the order in which vertices should be rendered. It takes the primitive type, the number of indices, the data type of the indices, and the offset to the indices as parameters.
8. **glUniform()**: This function is used to set uniform variables in shaders. It allows you to pass values from your application to the shader program, such as matrices, textures, or other parameters.
9. **glViewport()**: This function sets the viewport dimensions, specifying the area of the window where rendering will occur. It takes the x and y positions of the lower-left corner and the width and height of the viewport as parameters.
10. **glClearColor()**: This function sets the clear color for the color buffer. It takes the red, green, blue, and alpha components as parameters.

The perspective and transformation functions from the glm library are:

1. **glm::mat4**: This is a data type provided by the glm library representing a 4x4 matrix. It is commonly used to represent transformation matrices such as model, view, and projection matrices.
2. **glm::vec3**: This is a data type representing a 3D vector provided by the glm library. It is used to represent positions, directions, and other 3D coordinates.
3. **glm::translate()**: This function creates a translation matrix that can be used to move objects in 3D space. It takes a vector as a parameter specifying the translation offset.
4. **glm::rotate()**: This function creates a rotation matrix that can be used to rotate objects around a specific axis. It takes an angle in radians and a vector representing the rotation axis as parameters.
5. **glm::scale()**: This function creates a scaling matrix that can be used to scale objects in 3D space. It takes a vector as a parameter specifying the scaling factors along each axis.
6. **glm::perspective()**: This function creates a perspective projection matrix that simulates a perspective view in 3D space. It takes parameters such as the field of view, aspect ratio, and near/far clipping planes.
7. **glm::lookAt()**: This function creates a view matrix that positions the camera in the scene and determines the view direction. It takes parameters for the eye position, target position, and up vector.
8. **glm::mat4::identity()**: This function returns an identity matrix, which is the default state for transformation matrices.

These are just a few of the functions provided by the glm library that are commonly used in OpenGL to handle mathematical calculations and transformations.

The functions that I used to apply textures to the 3d objects are:

1. **glGenTextures()**: This function generates texture names (IDs) that can be used to reference textures in OpenGL.
2. **glBindTexture()**: This function binds a texture to a specific texture target, such as **GL\_TEXTURE\_2D**, so that subsequent texture operations affect the bound texture.
3. **glTexImage2D()**: This function specifies the image data for the texture. It uploads pixel data, such as the width, height, format, and pixel data itself, to the currently bound texture target.
4. **glTexParameteri()**: This function sets texture parameters, such as wrapping behavior, minification, and magnification filters, to control how the texture is sampled.
5. **glActiveTexture()**: This function selects the active texture unit. It is used when working with multiple textures.
6. **glUniform1i()**: This function sets the value of a sampler uniform in the shader program to specify the texture unit to be used for a specific texture.

Undoubtedly, the functions mentioned above represent just a fraction of the wide range of functions utilized throughout the process of rendering a 3D scene in OpenGL. However, I firmly believe that these functions serve as the core fundamentals, demonstrating my understanding and proficiency in this field. As OpenGL operates as a state machine, the potential to delve into a comprehensive analysis of its extensive function library seems boundless.

In a comprehensive paper, I could delve into the intricate details of various other vital functions employed during the rendering pipeline. These functions encompass vertex and fragment shaders, matrix transformations, lighting calculations, and more. The vertex shader processes vertices' positions, normals, and texture coordinates, enabling geometric transformations and manipulations. Meanwhile, the fragment shader determines the color and other attributes of each pixel, allowing for advanced lighting models and effects.

Furthermore, functions responsible for creating and managing the OpenGL context, such as **glfwCreateWindow()** and **glfwMakeContextCurrent()**, establish the foundation for rendering the scene. These functions handle window creation, OpenGL context initialization, and context activation, providing a vital framework for subsequent operations.

In addition to the functions mentioned earlier, a thorough understanding of other critical OpenGL functions, including **glDrawElements()**, **glViewport()**, **glClearColor()**, and **glEnable()**, is necessary. These functions enable efficient rendering of complex geometry, define the viewport's dimensions and transformation, specify the background color, and enable specific capabilities, respectively.

Furthermore, functions related to texture loading, such as **stbi\_load()** (from the stb\_image library), enable the retrieval of image data, while **glGenTextures()** and **glTexImage2D()** facilitate the creation and initialization of texture objects. The utilization of these functions empowers the application of captivating textures onto 3D objects, enhancing the visual appeal of the scene.

As I explore and expand upon these functions, their implementation details, and their interdependencies within the rendering pipeline, my paper could reach substantial lengths. Each function has its own intricacies, parameters, and impact on the overall rendering process. By dissecting and comprehensively explaining these functions, I can demonstrate my profound understanding of OpenGL's function library, solidifying the knowledge and skills I have acquired.

In conclusion, while the functions discussed above represent a foundational selection, the realm of OpenGL functions is vast and offers limitless opportunities for exploration and analysis. By exploring additional functions, such as those related to shaders, matrix transformations, and rendering optimizations, I can further expand the depth and breadth of my paper, showcasing my comprehensive understanding of OpenGL and its underlying principles. (*Learn OpenGL, Extensive Tutorial Resource for Learning Modern OpenGL*, n.d.)

***References***

*Learn OpenGL, extensive tutorial resource for learning Modern OpenGL*. (n.d.). <https://learnopengl.com/>

*Learn C++*. (n.d.). https://www.learncpp.com/

Overvoorde, A. (n.d.). *OpenGL - Introduction*. https://open.gl/