Computer Graphics

Leiria December 2023

Ricardo Brites 2202225

Table of Contents

OBJ Loading

The loading process of the OBJ model begins by opening the OBJ file using the ifstream library. Once the file is successfully opened, the program reads through its contents, extracting crucial data required for the model to be loaded. The pertinent information includes the locations of vertices, their normals, and the index orders that define the faces of the model.

This data, extracted during runtime, is then stored in a variable within the program. The comprehensive storage of vertices, normals, and index orders facilitates the subsequent rendering and manipulation of the 3D model within the game engine. This approach ensures that the necessary model information is readily available in memory for efficient use during runtime.

Shaders

Vertex Shader

The shader in question processes input attributes, including position, color, and normal, and performs transformations on the vertices. This involves applying the model, view, and projection matrices to modify the position of each vertex. The output of this vertex shader includes the vertex color, vertex normal, and the direction of light, which are then passed to the fragment shader for further processing.

Here's a breakdown of the shader functionality:

Inputs
- Position: Coordinates of the vertex.
- Color: Color information associated with the vertex.
- Normal: The normal vector representing the direction perpendicular to the surface at the vertex.

Transformations
- The shader applies the model, view, and projection matrices to transform the vertex position. These matrices are crucial for mapping the 3D model from its local space (model space) to the camera's view (view space) and then projecting it onto the 2D screen (clip space).

Outputs
- Vertex Color: The color information associated with the vertex.
- Vertex Normal: The normal vector after transformation.
- Light Direction: The direction of light, often used for lighting calculations in the fragment shader.

This vertex shader prepares the necessary data for subsequent fragment shader operations, such as shading and coloring, by transforming vertices and providing relevant information to the fragment shader.

Fragment Shader

The fragment shader processes information received from the vertex shader—specifically, the vertex color, vertex normal, and light direction. Its primary purpose is to compute lighting and determine the final colors of the (pixels) that contribute to the rendered image. Here's an overview of the fragment shader's functionality:

Inputs:
- Vertex Color: Color information associated with the vertex.
- Vertex Normal: The normal vector representing the direction perpendicular to the surface at the vertex.
- Light Direction: The direction of light, often calculated in the vertex shader.

Lighting Calculations

- Using the provided vertex color, vertex normal, and light direction, the fragment shader performs lighting calculations. The shader only calculates a diffuse map in this simple project but other maps would be used here to alter the appearance of the model.

Output

- The final color of the fragment is computed based on lighting calculations and is output as the color for that pixel in the rendered image.
- By taking into account the properties of the surface (vertex color and normal) and the lighting conditions (light direction), the fragment shader contributes to the realism and visual appeal of the rendered 3D model.

Camera Transformations (View Matrix)

The creation of the view matrix involves two key matrices: the rotation matrix and the location matrix. Both matrices are constructed with the assistance of the glm math library. The camera's location and rotation are updated as vectors, and these vectors are then transformed into matrices using glm. The view matrix is ultimately derived by multiplying the rotation matrix with the location matrix.

Here's a breakdown of the process:

Rotation Matrix

- The camera's rotation is updated as a vector, and this vector is converted into a rotation matrix using glm.

Location Matrix

- Similarly, the camera's location is updated as a vector, and this vector is converted into a location matrix using glm.

Multiplication

- The rotation matrix and the location matrix are multiplied together to obtain the final view matrix. This multiplication operation combines the rotational and translational aspects of the camera's transformation.

In summary, the view matrix is constructed by updating the camera's rotation and location vectors, converting these vectors into matrices using glm, and then multiplying the resulting matrices to obtain the combined transformation. The resulting view matrix is crucial for transforming objects from world space to camera space, providing the necessary information for rendering the scene from the perspective of the camera.

User Interaction

User interaction is integrated into the system through SDL events. The detection of specific events, such as key presses (W, A, S, D, Q, E) and mouse movement, triggers adjustments to the camera's rotation or location. Here's an overview of how user interaction is facilitated

Keyboard Input

W, A, S, D, Q, E Keys

- Detection of these keys initiates changes to the camera's location.
- For example, pressing W moves the camera forward, A and D control lateral movement, and E and Q trigger ascending and descending movement respectively.
- Mouse Input:

Mouse Movement

- Changes in the mouse position are detected to enable user control.
- Mouse movement is be used to adjust the camera's rotation, providing an intuitive way for users to navigate the virtual environment.

Handling Events

- SDL events are processed to identify when specific keys are pressed or when the mouse is moved.
- Depending on the detected event, corresponding adjustments are made to the camera's rotation or location.

By incorporating SDL events, the system creates a responsive and interactive user experience. Users can navigate and interact with the virtual environment using familiar input controls, enhancing engagement within the rendered scene.

Objectives

All outlined objectives have been successfully accomplished.

Bibliography

https://learnopengl.com/Getting-started/Shaders
https://www.khronos.org/opengl/wiki/Shader_Compilation#Shader_error_handling
https://www.youtube.com/@thecherno
https://www.youtube.com/watch?v=iCazI3nKBf0
https://stackoverflow.com/questions/8844585/glm-rotate-usage-in-opengl
https://github.com/g-truc/glm
https://registry.khronos.org/OpenGL-Refpages/gl4/html/glVertexAttribPointer.xhtml