**3D Model Viewer Program Documentation**

**Overview**

This project is a 3D Model Viewer implemented using OpenGL, GLEW, GLFW, and Assimp. It allows loading and rendering 3D models in formats such as OBJ and STL, providing basic camera controls for viewing the model. The program is modular and structured to facilitate further development.

**Project Structure**

**1. Main Program (main.cpp)**

* **Purpose**: Initializes the application, loads the 3D model, and manages the main rendering loop.
* **Key Responsibilities**:
  + Initialize GLFW and create the window.
  + Initialize GLEW for managing OpenGL extensions.
  + Load the 3D model using Assimp.
  + Set up the camera and shaders.
  + Handle the main loop for rendering and event processing.
  + Clean up resources on exit.

**2. Shader Management (shader.cpp & shader.h)**

* **Purpose**: Manages the creation, compilation, and linking of shaders used for rendering.
* **Key Responsibilities**:
  + **createShader()**: Compiles individual shaders (vertex/fragment) from source code.
  + **createShaderProgram()**: Links vertex and fragment shaders into a shader program.
* **Shader Source**:
  + **Vertex Shader**: Handles transforming vertices from model space to screen space.
  + **Fragment Shader**: Handles the lighting calculations and final pixel color.

**3. Model Management (model.cpp & model.h)**

* **Purpose**: Loads and processes 3D models using the Assimp library.
* **Key Responsibilities**:
  + **loadModel()**: Loads a model from a file and processes its nodes and meshes.
  + **processNode()**: Recursively processes each node in the model’s scene graph.
  + **processMesh()**: Extracts vertex data and indices from each mesh for rendering.
  + **computeBoundingBox()**: Computes the model's bounding box for camera positioning.

**4. Camera Management (camera.cpp & camera.h)**

* **Purpose**: Handles the camera’s position, orientation, and movement within the 3D scene.
* **Key Responsibilities**:
  + **Camera Struct**: Represents the camera’s properties like position, target, and orientation.
  + **updateCameraVectors()**: Updates the camera’s front, right, and up vectors based on yaw and pitch.
  + **panCamera()**: Moves the camera's position and target for panning.

**5. Callback Management (callbacks.cpp & callbacks.h)**

* **Purpose**: Manages user input (mouse, keyboard, and scroll) and updates the camera accordingly.
* **Key Responsibilities**:
  + **mouse\_callback()**: Handles mouse movement to rotate the camera.
  + **scroll\_callback()**: Handles scroll input to zoom the camera in and out.
  + **key\_callback()**: Handles keyboard input to reset the camera or other key events.
  + **renderScene()**: Handles rendering the scene each frame, including setting shader uniforms and drawing the model.

**Dependencies**

**Libraries Used:**

* **OpenGL**: Core graphics API for rendering.
* **GLEW**: Manages OpenGL extensions.
* **GLFW**: Handles window creation, input, and context management.
* **Assimp**: Loads and processes 3D model files.
* **TinyFileDialogs**: Provides file dialogs for selecting model files.

**Installation:**

Ensure all libraries are installed and linked correctly in your build environment. For example, if using vcpkg:

bash

Copy code

vcpkg install glew glfw3 assimp tinyfiledialogs

**Key Concepts and Usage**

**Loading Models:**

* Models are loaded using the loadModel() function. This function utilizes Assimp to read the model file, then processes each mesh to extract vertex and index data for rendering.

**Shader Management:**

* Shaders are crucial for rendering the model. The vertex shader transforms vertex positions from model space to screen space, while the fragment shader handles lighting and color.
* The createShaderProgram() function links the shaders and prepares them for use.

**Camera Controls:**

* The camera can be controlled using the mouse and scroll wheel:
  + **Left Mouse Button**: Rotates the camera around the model.
  + **Right Mouse Button**: Pans the camera.
  + **Scroll Wheel**: Zooms in and out.
  + **Key 'R'**: Resets the camera to its default position.

**Event Handling and Callbacks:**

* Input events are handled via GLFW callbacks:
  + **mouse\_callback()**: Updates the camera orientation based on mouse movement.
  + **scroll\_callback()**: Adjusts the camera's field of view (zoom) based on scroll input.
  + **key\_callback()**: Handles keyboard inputs for actions like resetting the camera.

**Rendering Loop:**

* The main rendering loop (while (!glfwWindowShouldClose(window))) handles:
  + Clearing the screen.
  + Setting up the view and projection matrices.
  + Drawing the model.
  + Swapping buffers and polling for events.

**Extending the Program**

**Adding More Shaders:**

* To add more shaders (e.g., for different rendering techniques), create new shader source files and manage them using the createShader() and createShaderProgram() functions.

**Supporting More Model Formats:**

* Assimp supports a wide range of model formats. To support additional formats, ensure the correct Assimp flags are used in loadModel().

**Improving Camera Controls:**

* You can extend the camera controls to support features like orbiting around the model, changing the camera type (e.g., orthographic), or adding animation paths.

**UI Enhancements:**

* Integrate a simple GUI (e.g., using ImGui) for features like loading new models, adjusting lighting, or changing shaders.

**Troubleshooting**

**Common Issues:**

1. **GLEW Initialization Failure**:
   * Ensure GLEW is initialized after the OpenGL context is created (glfwMakeContextCurrent(window)).
2. **Model Not Loading**:
   * Verify the file format is supported by Assimp and the correct path is provided.
3. **Camera Pointer is Null**:
   * Ensure glfwSetWindowUserPointer(window, &camera); is called before setting up the callbacks.
4. **Incorrect Model Orientation**:
   * Adjust the model matrix in the shader or preprocess the model to correct orientation issues.

**Final Notes**

This program is designed to be a starting point for more complex 3D model viewers or graphics applications. The modular structure allows developers to easily extend or replace components such as shaders, camera controls, or model loaders. Always ensure the libraries are up-to-date and compatible with your development environment.