**Reflection on Development Choices for the 3D Scene:**

When crafting the 3D scene, a series of deliberate choices were made to create an immersive and visually captivating environment. The selection of objects for the scene, including a conical frustum, a cuboid, a cylinder, and a sphere, was driven by the goal of showcasing an array of geometric shapes with unique properties. This approach allows for a comprehensive display of the application's rendering capabilities and provides a diverse visual experience for users.

The conical frustum was specifically chosen to highlight intricate geometry and shading effects. Its inclined sides and varying radii enable the demonstration of nuanced lighting effects and contribute to the creation of a sense of depth within the scene. The cuboid, on the other hand, serves as a representation of simplicity, allowing for the effective demonstration of texture mapping techniques. The cylinder's inclusion in the scene provides an opportunity to exhibit the rendering of curved surfaces and the seamless transition of shading across its form. Lastly, the sphere, being a foundational geometric shape, showcases the rendering of smooth surfaces, emphasizing the application's ability to accurately represent simple yet elegant forms.

In terms of the required functionality, considerable effort was invested in programming the scene to enable users to navigate and explore the virtual environment using a dynamic virtual camera. This functionality significantly enhances user engagement by providing them with the freedom to interact with the scene from different perspectives and angles. The implementation of the camera class was instrumental in achieving this, as it facilitates the management of camera movement and orientation through user input.

**User Navigation in the 3D Scene:**

User navigation within the 3D scene is primarily facilitated by the virtual camera control, which enables seamless movement and orientation adjustments.

**Camera Movement:** Users can utilize keyboard inputs to control the camera's movement. Keys such as "W," "A," "S," and "D" control forward, left, backward, and right movements, while the "Q" and "E" keys facilitate vertical movement, allowing users to ascend and descend within the scene.

**Camera Rotation:** The camera's orientation can be adjusted through mouse inputs. Mouse movements around the vertical axis (yaw) and the horizontal axis (pitch) alter the camera's viewing angle. The recalculation of the camera's orientation after each rotation ensures accurate movement and an uninterrupted exploration experience.

**Additional Controls:** To further enhance user experience and interaction, the escape key has been assigned the function of closing the application. Additionally, pressing the F11 key instantly changes the perspective to the User Navigation section, providing users with a quick way to jump to the navigation instructions.

These navigation controls are encapsulated within the camera class, enhancing the code's modularity and reusability while providing a user-friendly way for users to engage with the 3D scene.

**Custom Functions for Modularity and Organization:**

The development of the code was guided by a commitment to modularity and organization, which is evident through the creation of custom functions.

**Camera::MoveCamera():** This function encapsulates the logic for camera movement based on the direction and movement amount provided. Its existence simplifies the codebase by centralizing movement calculations and enhancing code readability.

**Camera::RotateBy():** Responsible for updating the camera's yaw and pitch angles in response to user input, this function ensures a consistent and controlled camera rotation. The clamping of the pitch angle guarantees that the camera's orientation remains within reasonable bounds.

**Camera::recalculateVectors():** By recalculating the camera's forward, right, and up vectors based on the current yaw and pitch angles, this function maintains the camera's correct orientation. Its implementation abstracts complex vector calculations and keeps the codebase organized.

**Shader::load():** The load function streamlines the process of shader compilation, program linking, and error handling. It takes vertex and fragment shader source code as inputs, simplifying the shader loading process and promoting a clean separation of concerns.

**Texture::Texture():** This constructor simplifies the process of loading image files as textures and creating OpenGL textures. Its existence enhances code reusability and improves readability, as the complexity of texture loading is abstracted.

These custom functions contribute to code maintainability and reusability, fostering an organized and efficient structure that aligns with best practices in software development. Their implementation is crucial to achieving a clean, modular, and comprehensible codebase.