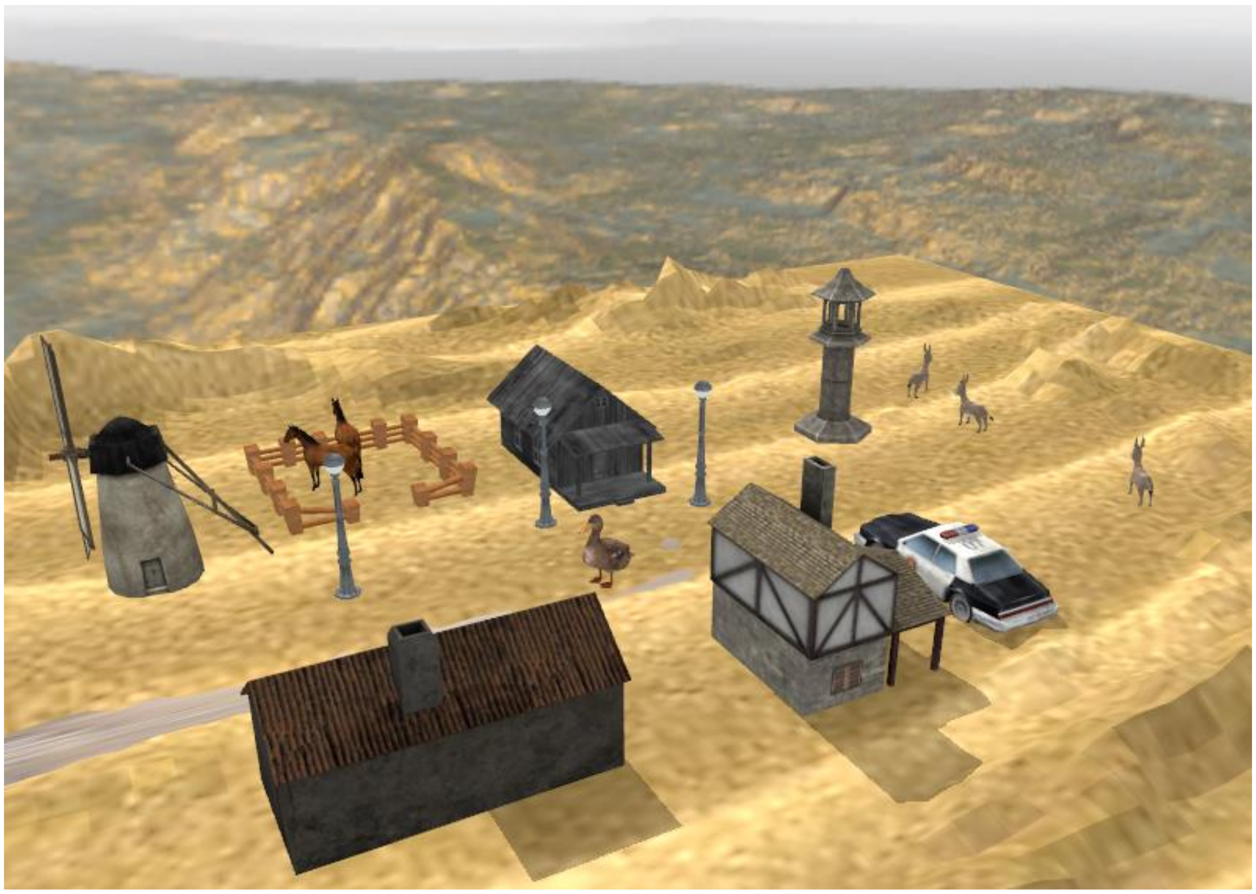
Graphic Processing

Project Documentation



Student: Sîrca Florentina Raluca

Group: 30433

Section: Computer Science

2023-2024

Contents

1. Subject specification

This project is about making realistic 3D pictures using the OpenGL library, and it specifically highlights a village in the western part. OpenGL is a common set of rules that helps with creating 2D and 3D parts in computer programs, and it works on different platforms. The scene I'm showing is of a small village in the western area, with various landscapes like mountains, hills and a desert. Users have the ability to interact directly with the scene using mouse and keyboard inputs, enabling them to navigate and manipulate objects within the environment.

2. Scenario

2.1. Scene and objects description

Within our scene, a collection of objects is arranged to form the overall setting. Additionally, there are standalone items that will be introduced individually to add animation to the scene.

To start, the scene includes a ground plane representing the surface of the desert, featuring an appropriate texture. The village is set against this desert backdrop. The various objects in the village include houses, a police car, a light tower, street lamps, a mill, and a big door to enter the village.

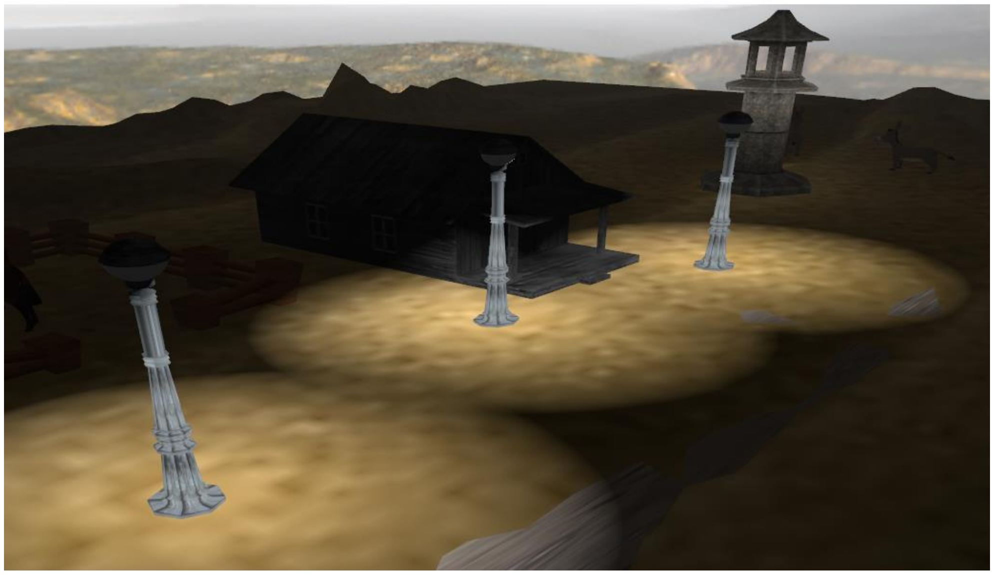
Adding a touch of life to the scene, we also feature animals like horses and donkeys, enhancing the village atmosphere with their presence. Each of these elements collectively contributes to crafting the ambiance of the western village within the scene.

2.2. Functionalities

In my western village scene, the lighting is powered by directional light, point light, and spot light sources. The bird is mobile along the scene on the ox axes, providing a dynamic element. The fans operate autonomously, adding movement while observing the haunted village.

Exploring the scene is made easy for the user; they can take a tour using keyboard keys or the mouse. The viewing experience can be customized by switching between solid view and wireframe, or choosing between polygonal and smooth face modes with simple keyboard commands. Fog effects can also be introduced at the user's discretion by adjusting the density.

For an atmospheric touch, all light sources can be toggled on or off by pressing corresponding keys, plunging the village into night. Additionally, users can opt to witness a rare desert phenomenon – rain – at their discretion, adding a unique and captivating element to the scene.



3. Implementation details

3.1. Functions and special algorithms

3.1.1. Possible solutions

Point lights are light sources with a specific position that radiantly and uniformly illuminate in all directions. Unlike directional lights, the rays generated by point lights fade with distance, causing objects closer to the source to appear more illuminated than those farther away. When using point lights, the light direction is not constant between fragments and must be calculated for each different fragment.

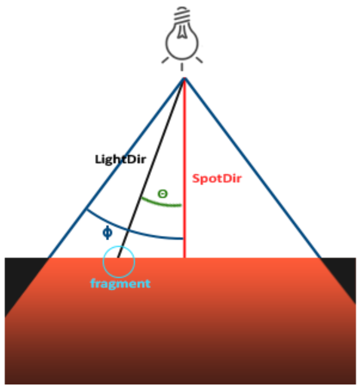
To diminish the intensity of point lights over distance, an attenuation coefficient must be applied. Attenuation can be calculated as a linear or quadratic function dependent on distance. Linear attenuation requires a less complex calculation but offers less realism because, in the real world, light sources are bright when close to objects but quickly diminish in brightness as we move farther away. Realism is significantly enhanced when incorporating quadratic attenuation based on distance. Thus, attenuation can be calculated using:

$$Att = \frac{1.0}{\mathrm{Kc} + \mathrm{Kl} * \mathrm{d} + \mathrm{Kq} * \mathrm{d2}}$$

where Kc is a constant term, usually held at 1.0, Kl is a linear term, Kq is a quadratic term, and d is the distance. **(Lab Guide - Graphics Processing)**

A spotlight or street lamp is a light source situated in the surrounding environment that, instead of emitting light rays in all directions, projects them only in a specific direction. Consequently, only objects within a certain range of the spotlight's direction are illuminated, leaving the rest in darkness. A prime example of a reflector would be a street lamp or a lantern.

In OpenGL, a spotlight is defined by its position, propagation direction, and a cutoff angle specifying the range of the spotlight. For each fragment, we calculate whether it lies within the cutoff directions of the spotlight (i.e., within its cone), and if so, we color the fragment accordingly. The image below provides a graphical representation of how a spotlight operates **(Vries, 2020).**



The fog effect is achieved by manipulating attributes associated with the fog effect, allowing us to customize the atmosphere and enhance the perception of depth (Z-axis). The background color should be chosen based on the color of the fog effect.

fogColor = vec4(0.5f, 0.5f, 0.5f, 1.0f);

In the end, the color is calculated based on the density of the fog and combined with the light in the scene.

fColor = mix(fogColor, vec4(color, 1.0f), fogFactor);

where $fogFactor = e^{-(fragmentDistance * fogDensity)2}$

The sky object is implemented in the form of a skybox, similar to the one in the laboratory.

For the rain effect, I opted for loading an object resembling a raindrop, which I duplicated multiple times to simulate rainfall. The raining function handles the logic for updating the positions of two raindrops (r1 and r2) as they fall vertically. If a raindrop reaches a certain threshold, it is reset to a higher position, creating a continuous rain effect. The renderRain function, likely called to render the raindrops, utilizes a specified shader. It invokes the raining function to update raindrop positions and sets up the model matrix for rendering. The raindrops are drawn twice, one for each raindrop. The code includes translations to determine the raindrop positions and integrates with a shader to render the raindrops dynamically, creating a visual representation of falling rain in the 3D scene.

Shadow Mapping is a multi-pass technique employed in computer graphics to enhance the realism of OpenGL scenes by simulating the effects of shadows. While conventional lighting techniques effectively illuminate objects, they often neglect the impact of occlusions, resulting in scenes devoid of realistic shadows. Shadows play a crucial role in mimicking real-world scenarios, contributing to improved depth perception.

The key principle behind shadow mapping involves viewing the scene from the perspective of the light source rather than the final camera viewpoint. This shift in perspective allows the identification of areas in shadow, where objects are not directly illuminated due to obstructing elements.

Capture the scene from the viewpoint of the light source, focusing solely on depth values. These values are stored in a shadow map or depth map, utilizing a framebuffer object and depth texture. This pass establishes a direct correlation between depth values and their spatial representation in the scene.

Render the scene from the final camera position, comparing the depth of each visible fragment with the depth values stored in the shadow map. Fragments with greater depth than the corresponding values in the depth map are considered in shadow.

The initial pass of the shadow mapping algorithm involves creating a depth map of the entire scene from the light's perspective. This depth map is directly stored in a texture, configured through a framebuffer object. Key steps include creating a framebuffer object, generating a depth texture, and attaching it to the framebuffer. Additional considerations such as texture parameters, size, and proper attachment finalization contribute to the successful generation of the depth map.

Transforming the scene to the light's space is crucial for the rendering process from the light's perspective. Utilizing a directional light for this laboratory, transformations involve creating a view matrix using the light's direction and employing an orthographic projection to prevent perspective distortion. The resulting transformation matrix, termed the light space transformation matrix, is pivotal for subsequent rendering stages.

The vertex shader is responsible for transforming all vertices into the light's space, ensuring proper alignment with the light's perspective. A simple fragment shader, generating dummy output, accompanies the vertex shader. The rendering process to the depth buffer involves configuring shaders, setting the viewport, and rendering the scene, effectively populating the depth map.

The fragment shader plays a crucial role in shadow computation, modulating the final color of each fragment based on the computed shadow value. The process involves transforming the fragment's position into normalized coordinates, performing perspective divide, and comparing the fragment's depth with the closest depth value from the light's perspective. This comparison determines whether the fragment is in shadow or illuminated, contributing to the final visual output.

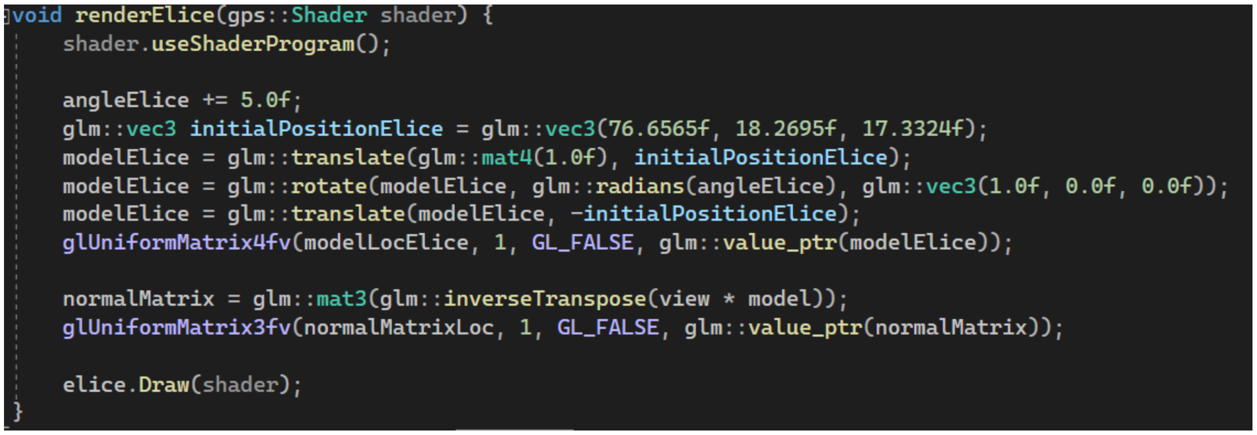
3.1.2. The motivation of the chosen approach

In the labs focused on this topic, we acquired the knowledge of computing light, shadows, and applying translation, scaling, and rotation operations to various objects. This foundation has proven invaluable as it equipped us with essential skills to integrate these functionalities into my project.

3.2. Graphics model

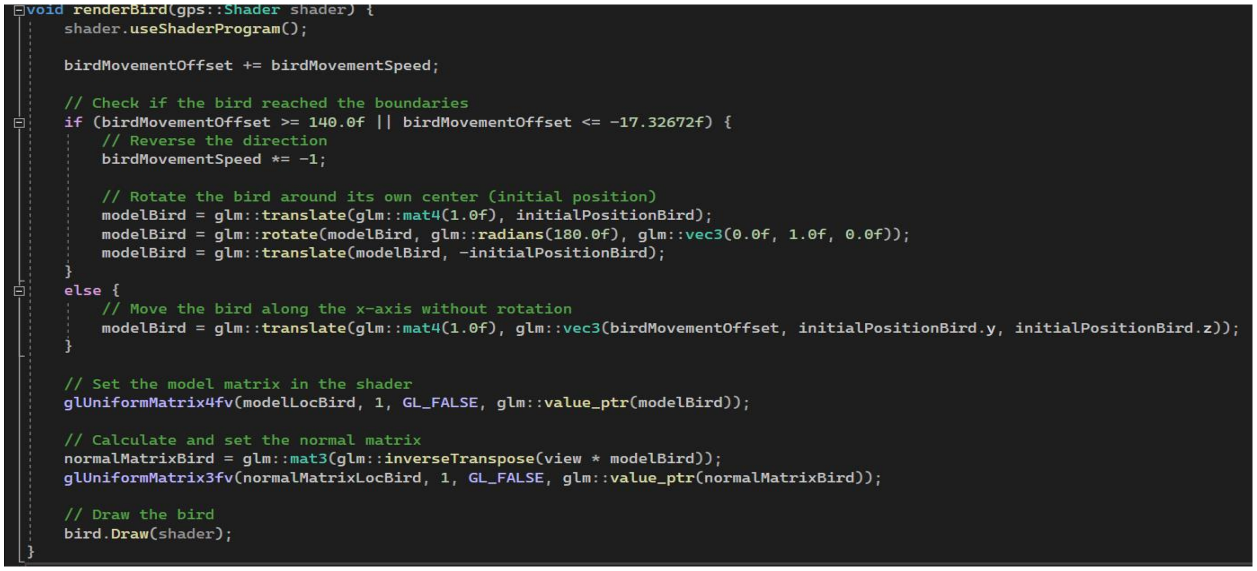
As for the models used, they are divided into the actual scene, referred to as "scene," and separate objects that will be animated. Thus, we can distinguish the following graphic models:

- Scene: The actual scene, which will be drawn simply without modifications.
- Fans: The wind turbine blades in the scene; these will have a continuous rotation animation around the X-axis. To achieve this rotation correctly, the object is translated to the origin, rotated, and then placed back to its original position.



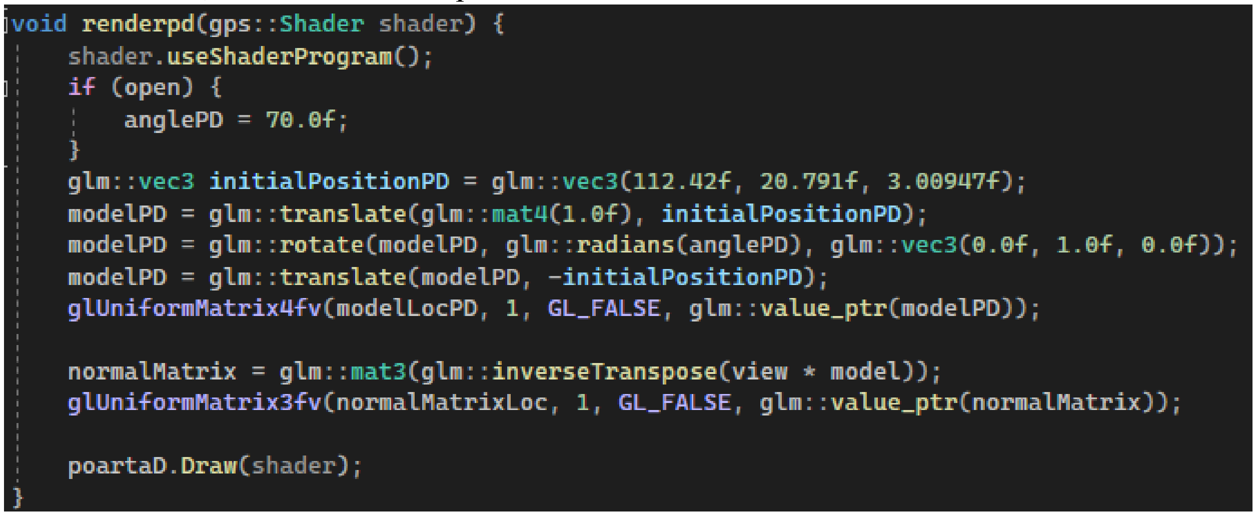
```
void renderElice(gps::Shader shader) {
    shader.useShaderProgram();

    angleElice += 5.0f;
    glm::vec3 initialPositionElice = glm::vec3(76.6565f, 18.2695f, 17.3324f);
    modelElice = glm::translate(glm::mat4(1.0f), initialPositionElice);
    modelElice = glm::rotate(modelElice, glm::radians(angleElice), glm::vec3(1.0f, 0.0f, 0.0f));
    modelElice = glm::translate(modelElice, -initialPositionElice);
    glUniformMatrix4fv(modelLocElice, 1, GL_FALSE, glm::value_ptr(modelElice));

    normalMatrix = glm::mat3(glm::inverseTranspose(view * model));
    glUniformMatrix3fv(normalMatrixLoc, 1, GL_FALSE, glm::value_ptr(normalMatrix));

    elice.Draw(shader);
}
```

- Bird: The function checks if the bird has reached predefined boundaries, and if so, it reverses its movement direction and rotates the bird 180 degrees around its initial position. This creates a smooth back-and-forth motion effect for the bird within the specified boundaries. When the bird is within the boundaries, it moves along the x-axis without rotation. The model and normal matrices are updated accordingly and sent to the shader to achieve the proper positioning and orientation of the bird in the scene. Ultimately, the function draws the bird using the specified shader, contributing to the animated dynamics of the overall 3D environment.



```
void renderBird(gps::Shader shader) {
    shader.useShaderProgram();

    birdMovementOffset += birdMovementSpeed;

    // Check if the bird reached the boundaries
    if (birdMovementOffset >= 140.0f || birdMovementOffset <= -17.32672f) {
        // Reverse the direction
        birdMovementSpeed *= -1;

        // Rotate the bird around its own center (initial position)
        modelBird = glm::translate(glm::mat4(1.0f), initialPositionBird);
        modelBird = glm::rotate(modelBird, glm::radians(180.0f), glm::vec3(0.0f, 1.0f, 0.0f));
        modelBird = glm::translate(modelBird, -initialPositionBird);
    }
    else {
        // Move the bird along the x-axis without rotation
        modelBird = glm::translate(glm::mat4(1.0f), glm::vec3(birdMovementOffset, initialPositionBird.y, initialPositionBird.z));
    }

    // Set the model matrix in the shader
    glUniformMatrix4fv(modelLocBird, 1, GL_FALSE, glm::value_ptr(modelBird));

    // Calculate and set the normal matrix
    normalMatrixBird = glm::mat3(glm::inverseTranspose(view * modelBird));
    glUniformMatrix3fv(normalMatrixLocBird, 1, GL_FALSE, glm::value_ptr(normalMatrixBird));

    // Draw the bird
    bird.Draw(shader);
}
```

- Door: The function checks if the door is open, and if so, it sets the rotation angle to 70 degrees, indicating an open door state. The initial position of the door is defined, and a model transformation matrix (modelPD) is created to position and rotate the door accordingly. The resulting model matrix is transmitted to the shader through a uniform variable (modelLocPD). Additionally, a normal matrix is calculated to handle lighting computations, considering the combined view and model matrices. This normal matrix is also sent to the shader via a uniform variable (normalMatrixLoc). Finally, the door is drawn using the specified shader, encapsulated in the poartaD.Draw(shader) call. Overall, the function manages the rendering aspects, including state checks, transformations, and shader communication, for a door in the OpenGL scene.



```
void renderpd(gps::Shader shader) {
    shader.useShaderProgram();
    if (open) {
        anglePD = 70.0f;
    }
    glm::vec3 initialPositionPD = glm::vec3(112.42f, 20.791f, 3.00947f);
    modelPD = glm::translate(glm::mat4(1.0f), initialPositionPD);
    modelPD = glm::rotate(modelPD, glm::radians(anglePD), glm::vec3(0.0f, 1.0f, 0.0f));
    modelPD = glm::translate(modelPD, -initialPositionPD);
    glUniformMatrix4fv(modelLocPD, 1, GL_FALSE, glm::value_ptr(modelPD));

    normalMatrix = glm::mat3(glm::inverseTranspose(view * model));
    glUniformMatrix3fv(normalMatrixLoc, 1, GL_FALSE, glm::value_ptr(normalMatrix));

    poartaD.Draw(shader);
}
```

3.3. Data structures

The project implementation utilized data structures from the GLM library for various data types, such as matrices and vectors, alongside a few custom-defined classes (Camera, Mesh, Shader, Window, Model3D).

3.4. Class hierarchy

The class hierarchy can be described as follows:

• Camera: a class encompassing all possible actions applicable to the camera;

• Main: a class comprising the main application;

• Window: a class containing operations necessary for opening the application as a window;

• Mesh: a class covering operations for loading textures for objects within the scene;

• Model3D: a class encompassing and executing all operations for loading objects within the scene.

4. Graphical user interface presentation / user manual

The user can interact with the virtual world with the following keys:

W – zoom in – move the camera forward

S – zoom out – move the camera backward

A – move the camera to the right

D – move the camera to the

J – move the light to the left

L – move the light to the right

1 – show wireframe view

2 – show point view

3 – show solid without smooth view

4 – show solid and smooth view

5 – start the preview

6 – stop the preview

7 – start the rain

8 – stop the rain

N – start the night mode, the lights are on

Z– stop the night mode, the lights are off

F – the density of fog is increasing and then decreasing

P – open the door

5. Conclusions and further developments

In conclusion, the Graphic Processing project has successfully implemented a realistic 3D representation of a western village using the OpenGL library. The project allows users to interact with the scene, navigate through the village, and manipulate various objects. The inclusion of diverse elements such as houses, vehicles, animals, and dynamic lighting sources contributes to creating a vibrant and immersive environment.

The implementation details showcase the use of OpenGL functionalities, including point lights, spotlights, fog effects, and dynamic elements like a moving bird and rotating fans. Special algorithms such as shadow mapping enhance the realism of the scenes by simulating the effects of shadows. The utilization of various data structures and a well-defined class hierarchy ensures an organized and efficient implementation.

Moving forward, there are several avenues for further development in this project. One aspect to explore is the enhancement of visual effects and realism, possibly by incorporating more advanced lighting techniques, texture mapping, or advanced shaders. Additionally, expanding the scene with more intricate details, diverse landscapes, or additional interactive elements could enrich the user experience.

Furthermore, optimizations in terms of performance and efficiency can be pursued. This may involve refining algorithms, implementing level-of-detail techniques, or exploring parallel processing to handle more complex scenes seamlessly.

Considering the user interface, improvements can be made to provide users with more intuitive controls and additional options for customizing their experience. The inclusion of a detailed user manual or on-screen instructions can enhance user engagement.

In summary, the Graphic Processing project has laid a solid foundation for creating visually compelling 3D scenes. Future developments can focus on pushing the boundaries of realism, optimizing performance, and refining user interactions to elevate the overall quality of the project.

6. References

[Reference 1] Lab Guide - Graphics Processing

[Reference 2] Vries, A. (2020). Advanced Graphics Programming with OpenGL. Apress.

[Reference 3] https://free3d.com
[Reference 4] https://www.rigmodels.com