Graphic Processing Project - OpenGL Application

Flavia-Maria Marincău
Group 30432

January 17, 2025

Contents

Chapter 1

Subject Specification

The primary objective of this project is to deepen our understanding of photorealism in games, but also to teach us how computers process a scene of objects. This project not only focuses on graphical rendering techniques but also helps us become proficient in using various 3D libraries and frameworks, such as OpenGL, GLFW, and GLM. The user of this application should have the ability to navigate the scene using both the keyboard and mouse, just like in a typical video game.

The theme of this project is centered around creating a 3D natural environment, featuring a house, trees, flowers, and a fountain. These objects are carefully arranged and textured to create a realistic and visually appealing environment. The textures were a challenging but rewarding part of the project, and they helped bring the scene to life with detailed surfaces and lifelike elements.

Through the development of this project, we aimed to better understand how 3D scenes are constructed and rendered by modern computers. The integration of user controls for navigating the scene, combined with the application of advanced lighting and texture techniques, demonstrates the potential for creating rich, interactive 3D environments.

The engine has been designed with performance in mind, ensuring that the scene is rendered efficiently without compromising visual quality.

Overall, this project aims to present a combination of cutting-edge 3D graphics techniques and a nature-themed environment that offers users a visually stunning and interactive experience. By integrating realistic lighting, textures, shadows, and user-controlled interactions, the project demonstrates the power of modern 3D graphics rendering in creating immersive virtual worlds. The result is a beautiful, interactive 3D scene where users can explore, interact, and enjoy a fully realized natural environment.

Chapter 2

Scenario

This project represents the design and implementation of an advanced 3D graphics application that focuses on rendering a highly detailed and immersive natural environment. The central theme of the project revolves around a nature scene, consisting of various elements like a house, a collection of trees, colorful flowers, and a fountain. These elements are modeled and textured to capture the beauty and simpless of a garden house-like setting.

2.1 Scene and Objects Description

The nature setting and its containing objects are encapsuled in a scene designed with Blender. Additional elements include the skybox and the light cube that represents a separate object, which can be visualised with the help of a movable camera to explore the environment.

- **Scene:** The main part of the environment, featuring a house, trees, flowers, and a fountain. These objects are arranged to create a natural, photorealistic setting. The scene is textured using detailed textures that simulate real-world materials and surfaces. Some objects in the scene have depth maps, roughness textures, height maps, metallic maps, and normal maps applied to them to enhance realism. These maps add detail and dynamic surface effects, improving lighting and shadow interaction, and creating a more lifelike appearance. Dynamic lighting and shadows are applied throughout the scene to enhance realism. The user can navigate the scene freely using keyboard and mouse controls, simulating a gaming environment.

To load the 3D model representing the scene, the following line of code is used in the project:

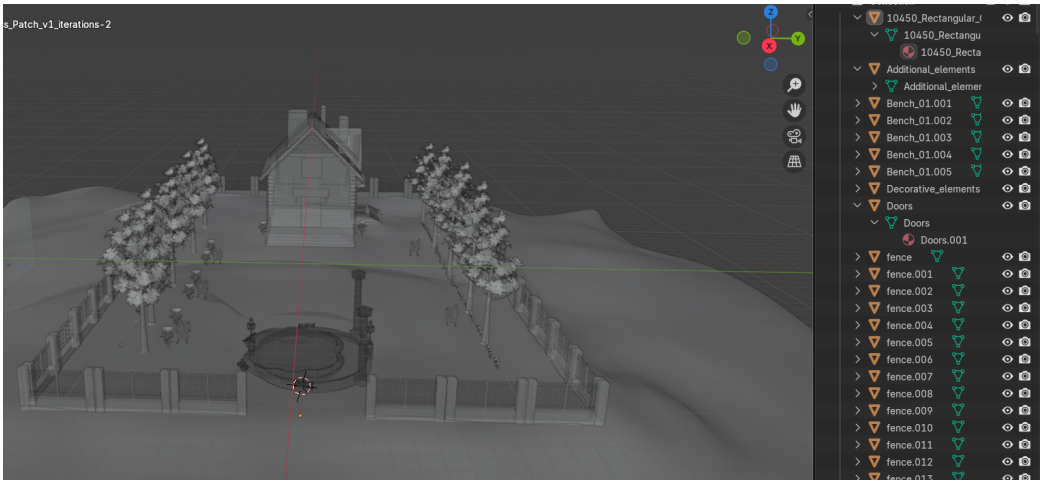

Figure 2.1: General scene (solid view) in Blender

To load the 3D model representing the scene, the following line of code is used in the project:

```
scene = gps::Model3D("objects/nature/nature.obj", "objects/nature/");
```

Here's a breakdown of the code:

- 'gps::Model3D': This refers to the class responsible for loading 3D models in the project. The 'gps' namespace suggests that it is part of a custom or third-party library designed for handling 3D objects.

- 'objects/nature/nature.obj': This is the path to the model file. The '.obj' file format is commonly used for 3D models and contains the geometry (vertices, edges, and faces) of the objects. It is the primary file that holds the 3D structure of the scene elements, such as the house, trees, and fountain.

- 'objects/nature/': This is the directory where the textures for the 3D model are stored. It ensures that the model's textures (such as surface colors, normals, and other material properties) are correctly loaded and applied. These textures are essential for enhancing the realism of the scene by providing detailed surface effects.

This line of code ensures that the model is loaded into the program, where it will be displayed in the scene with its textures and materials applied.

- **Cube Light:** A cube light is used to provide a primary light source in the scene. This light is crucial in illuminating the objects in the

environment and in casting realistic shadows. The light is adjustable, allowing the user to manipulate the intensity and direction, which impacts the appearance of shadows and highlights in the scene. This functionality allows for different lighting conditions to be explored.

Similarly, to load the 3D model representing the light source, the following line of code is used in the project:

```
lightCube = gps::Model3D("objects/cube/sun.obj", "objects/cube/");
```

- **Skybox:** The skybox creates the backdrop of the scene, providing a realistic horizon. It is composed of six textured images that form a cube surrounding the scene. The skybox simulates the sky, contributing to the sense of depth and scale. It serves as an environmental element that further enhances the overall realism of the scene.

  Below is a representation of how a skybox is loaded in an OpenGL Application.

```
faces.push_back("skybox/Sorsele33/bluecloud_ft.jpg");
faces.push_back("skybox/Sorsele33/bluecloud_bk.jpg");
faces.push_back("skybox/Sorsele33/bluecloud_up.jpg");
faces.push_back("skybox/Sorsele33/bluecloud_dn.jpg");
faces.push_back("skybox/Sorsele33/bluecloud_rt.jpg");
faces.push_back("skybox/Sorsele33/bluecloud_lf.jpg");
mySkyBox.Load(faces);
```

  The provided code snippet adds textures to a list, where each texture corresponds to one of the six faces of the skybox: front, back, up, down, right, and left. The textures, such as `bluecloud_ft.jpg`, `bluecloud_bk.jpg`, and so on, represent images that form the skybox, creating a 360-degree background environment for the scene. The `Load()` function of the `mySkyBox` object is then called to apply these textures to the skybox, rendering the skybox in the 3D scene as a realistic distant environment. This method helps provide an immersive experience by simulating a larger world surrounding the user, even though the scene itself may be limited to a smaller, more confined space.

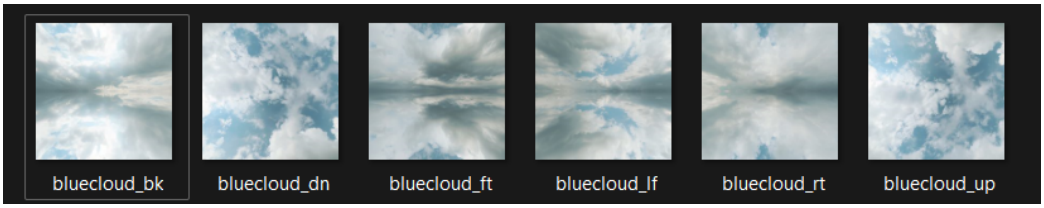

Figure 2.2: Skybox pictures

2.2 Functionalities

The project includes various interactive functionalities and graphical techniques designed to enhance the user experience and the realism of the scene. These are outlined as follows:

- **Easy movement of camera:** The camera can be moved using the arrow keys and mouse, providing a smooth navigation experience through the scene, similar to gameplay in 3D environments.
- **Control the speed of the camera:** The user can adjust the camera movement speed using the scroll wheel, offering flexibility in how quickly or slowly they navigate the scene.
- **Different lights implementation:** The scene features various light types, including:
  - **Directional light:** Generated by the cube light, providing global illumination across the scene.
  - **Spotlight:** Focused light that can be adjusted by the user to target specific areas.
  - **Flashlight:** A point light controlled via keyboard input, offering the ability to illuminate dark areas.
- **Shadow mapping technique:** A technique for casting realistic shadows in the scene, enhancing the depth and realism of the 3D objects.
- **Quality textures and maps:** The scene uses high-quality textures and surface maps, including:
  - Roughness maps
  - Normal maps

  - Height maps
  - Metallic maps

  These maps provide a more realistic appearance by simulating various material properties, improving the interaction of objects with light and shadows.

- **Fog effect:** The fog effect adds depth and atmosphere to the scene, and it can be adjusted by the user to create different visual moods and levels of visibility.

Chapter 3

Implementation Details

3.1 Functions and Special Algorithms

3.1.1 Possible Solutions

Several methods were considered to implement lighting and shadowing in the scene:

- Deferred Rendering: For complex lighting scenarios with multiple lights.
- Shadow Mapping: To efficiently implement shadows and lighting effects for static and dynamic objects.

3.1.2 The Motivation of the Chosen Approach

Shadow mapping was chosen due to its simple implementation in OpenGL and good performance in real-time rendering environments. This approach generates a depth map from the light's perspective, which is then used to determine whether a fragment is in shadow.

Point lighting was incorporated to simulate a flashlight effect. The user can toggle the flashlight on and off using keyboard input, adding interactivity to the scene.

3.1.3 Vertex Shader Implementation

In the vertex shader, we compute various outputs to handle lighting and shadows:

- `FragPos`: The world-space fragment position.

- `Normal`: The transformed normal vector for proper lighting calculations.
- `TexCoords`: Texture coordinates for the fragments.
- `FragPosLightSpace`: The fragment position transformed into the light's view space for shadow mapping.

These values are passed to the fragment shader for further processing.

3.1.4 Fragment Shader and Shadow Calculation

In the fragment shader, we implement the Blinn-Phong lighting model. We calculate whether a fragment is in shadow by comparing its depth to the depth stored in the shadow map. If the fragment is in shadow, the shadow value is set to 1.0, and the lighting components are multiplied by this value.

A small shadow bias is applied to avoid shadow acne, which can occur when fragments are incorrectly considered in shadow due to depth map resolution limitations.

3.1.5 Shadow Mapping Artifacts and Solutions

Several shadow mapping issues were encountered during implementation:

- **Shadow Acne**: Caused by fragments incorrectly being considered in shadow due to depth precision issues. This was solved using a shadow bias.
- **Over Sampling**: Regions outside the light's frustum incorrectly appear in shadow. This was resolved by clamping the texture coordinates outside the light's frustum to a default value of 1.0.

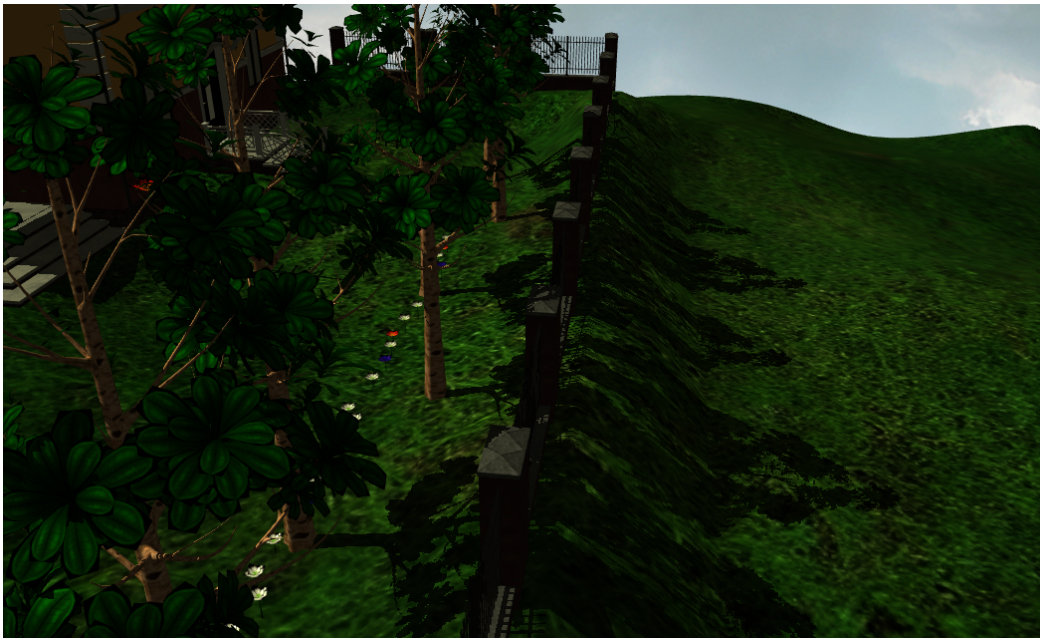

Figure 3.1: Shadow mapping example

3.1.6 Flashlight Implementation

To simulate a flashlight effect, we incorporated a point light that the user can toggle on and off using keyboard input. The flashlight's light source is modeled as a point light with varying intensity based on user interaction. This allows for dynamic lighting in the scene, where objects illuminated by the flashlight cast shadows and react to changes in lighting direction.

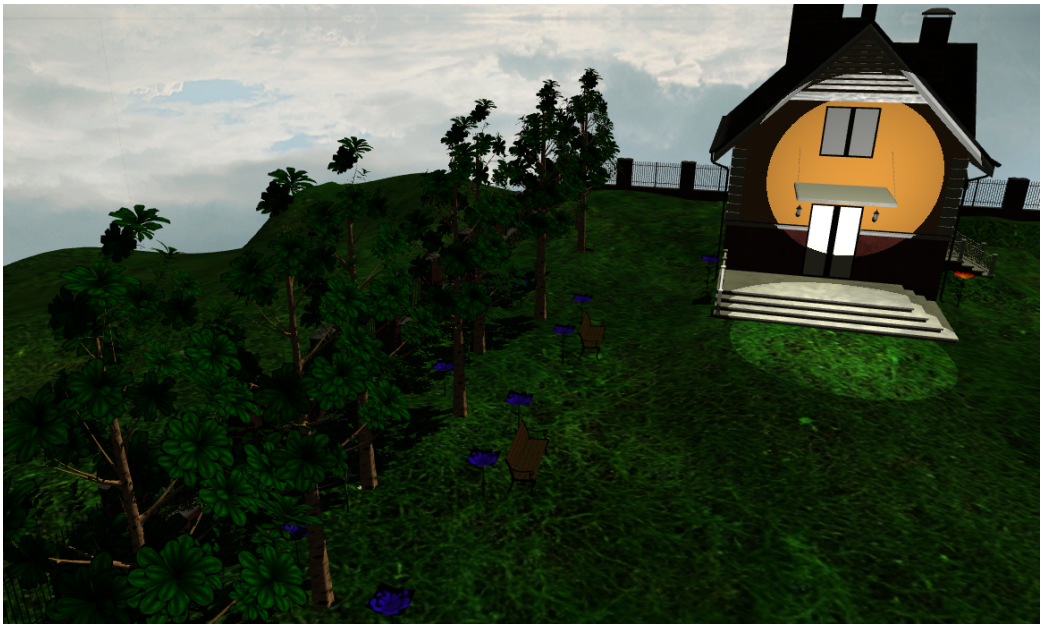

Figure 3.2: Flash light example

3.2 Graphics Model

The graphics model follows a forward rendering pipeline with shadow mapping to achieve realistic lighting and shadow effects. The main stages of the rendering process are as follows:

- **First pass**: In the first pass, the depth map is rendered from the light's point of view. This depth map is later used to compare fragment depths to determine whether they are in shadow.
- **Second pass**: During the second pass, the scene is rendered with proper lighting calculations (ambient, diffuse, and specular), and shadows are applied using the depth map from the first pass.
- **Point light toggle**: The point light is toggled on or off dynamically. When active, it is visually represented by a small cube rendered at the light's position.

A simplified version of the ‘renderScene‘ function illustrates the pipeline:

3.3 Graphics Model

The graphics model follows a forward rendering pipeline with shadow mapping to achieve realistic lighting and shadow effects. The main stages of the rendering process are as follows:

- **First pass**: In the first pass, the depth map is rendered from the light's point of view. This depth map is later used to compare fragment depths to determine whether they are in shadow.
- **Second pass**: During the second pass, the scene is rendered with proper lighting calculations (ambient, diffuse, and specular), and shadows are applied using the depth map from the first pass.

A simplified version of the 'renderScene' function illustrates the pipeline:

```
// first pass: render depth map from the light's point of view
depthMapShader.useShaderProgram();
glUniformMatrix4fv(glGetUniformLocation(depthMapShader.shaderProgram, "lightSpac
    1, GL_FALSE, glm::value_ptr(computeLightSpaceTrMatrix()));
glBindFramebuffer(GL_FRAMEBUFFER, shadowMapFBO);
glClear(GL_DEPTH_BUFFER_BIT);
scene.Draw(depthMapShader);

//second pass: render the scene with lighting and shadows
myCustomShader.useShaderProgram();
glUniformMatrix4fv(glGetUniformLocation(myCustomShader.shaderProgram, "lightSpac
    1, GL_FALSE, glm::value_ptr(computeLightSpaceTrMatrix()));
scene.Draw(myCustomShader);

//set flashlight uniforms
glUniform3fv(pointLightPosLoc, 1, glm::value_ptr(pointLightPos));
glUniform3fv(pointLightColorLoc, 1, glm::value_ptr(pointLightColor));
glUniform1f(pointLightRadiusLoc, pointLightRadius);
glUniform1i(pointLightEnabledLoc, pointLightEnabled);
```

In this function:

- The first pass renders the depth map using the light's view matrix and the computed light space transformation matrix.
- The second pass applies lighting and shadow effects using the depth map, along with the light space transformation matrix.

- The flashlight position and settings are updated dynamically based on the camera's position.

3.4 Data Structures

The following data structures are used in the implementation:

- **Scene**: Contains objects in the 3D space, including models, textures, and materials.
- **Camera**: Defines the user's viewpoint and allows movement through the scene. The Camera class requires several vectors, such as the camera position, the viewing direction, and the camera's position coordinates. These vectors are crucial for computing the view matrix and controlling the camera's movement within the scene.
- **Glint**: A structure that stores addresses of uniform variables in the shaders. It also contains vectors of different dimensions and sizes to manage shader variables efficiently.
- **Model**: A structure that holds information about 3D models, including vertex data, transformations, and material properties.
- **Shaders**: These structures represent the shaders used for rendering. They include program IDs, shader compilation, and linking data.
- **Primitive Types**: The implementation also uses standard C++ data types such as `bool`, `int`, and `char`. These are essential for implementing various functionalities, such as enabling/disabling features, looping over objects, and managing input/output.

Each of these data structures plays a critical role in the smooth operation of the graphics pipeline, enabling efficient rendering and interactive behavior within the scene.

3.5 Class Hierarchy

The project is structured around a set of interconnected classes, each designed to handle specific responsibilities within the graphics pipeline. Each class has its own header file, which encapsulates its properties and functions, ensuring modularity and ease of maintenance.

- **Main Application Class**: The main class of the application controls the flow of the entire program. It initializes the OpenGL context, manages user inputs, and oversees the rendering process. This class also handles window creation, input processing, and the main loop of the application. The main class coordinates the interaction between the different components of the system, such as the camera, scene, shaders, and lighting.
- **Camera Class**: This class is responsible for defining and managing the camera's properties, such as its position, view direction, and projection. It allows for the camera to move through the 3D space and adjust the view matrix dynamically based on user input. It contains methods for moving the camera, rotating it, and updating the view matrix, which is essential for rendering the scene from the correct perspective.
- **Model3D Class**: The Model class represents 3D models in the scene. It contains the vertex data, transformation matrices, and material properties of the model. Each model is composed of one or more meshes, and this class facilitates loading, transforming, and rendering those meshes.
- **Shader Class**: This class encapsulates the management of OpenGL shaders. It provides functions to compile shaders, link them into a program, and set uniform variables for rendering. The Shader class is used by other components (e.g., Scene, Model) to apply specific effects during the rendering process, such as lighting and shadow mapping.

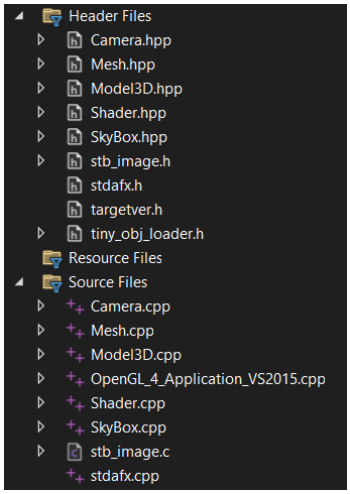

Figure 3.3: General overview of classes in Visual Studio

Chapter 4

Graphical User Interface Presentation / User Manual

The general view of the scene is presented below.

Figure 4.1: General aspect of the application

The user interface consists of the following controls:

- **W, A, S, D**: Move the camera forward, left, backward, and right, respectively.

- **Q, E**: Rotate the camera.
- **Mouse**: Look around and adjust the camera's orientation.
- **Y**: Toggle the point light (flashlight) on and off.
- **M**: Toggle wireframe mode for polygon rendering.

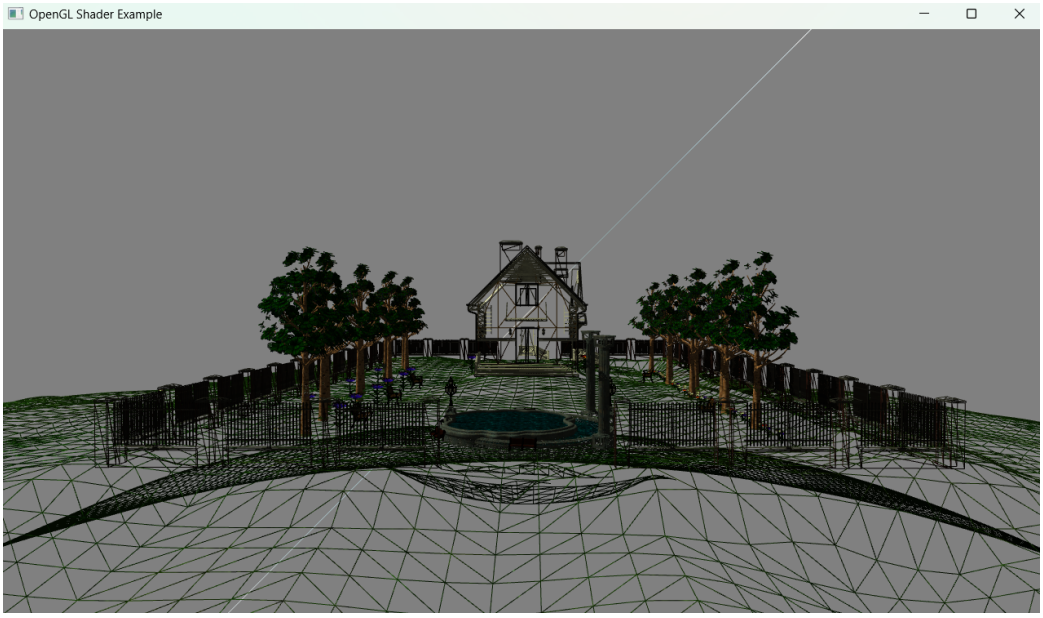

Figure 4.2: Wireframe rendering mode

- **N**: Toggle textured polygons (regular view).
- **V**: Toggle point rendering mode.

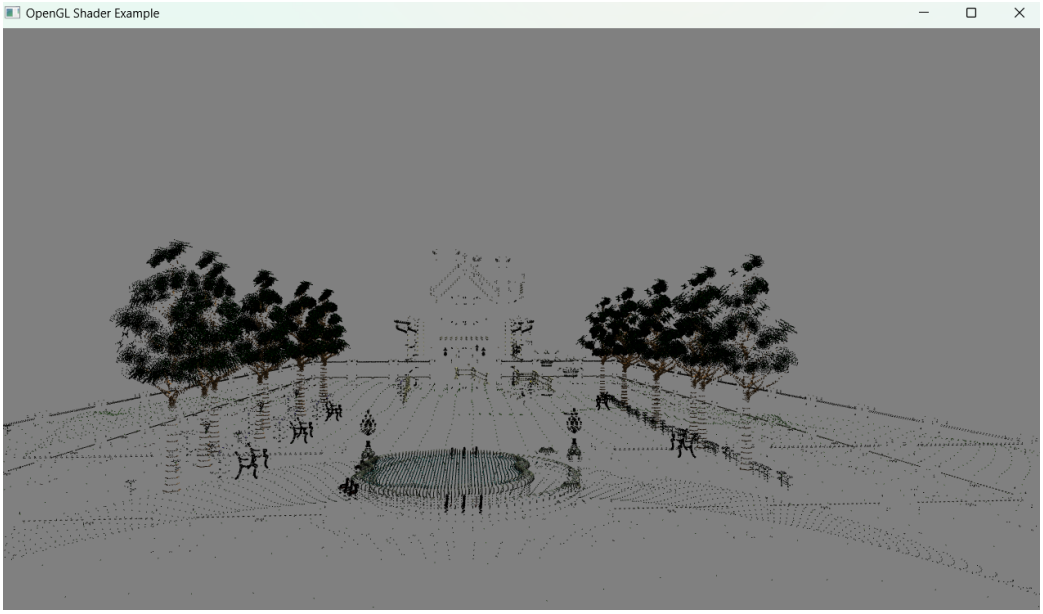

Figure 4.3: Point rendering mode

- **J**: Accelerate the sun's rotation.
- **L**: Stop the sun's rotation in a fixed direction.
- **1**: Adjust spotlight light direction.
- **Numpad +**: Increase light intensity.
- **Numpad -**: Decrease light intensity.
- **Space**: Play camera animation by reading directions from a file.
- **F**: Toggle fog effect on and off.

Figure 4.4: Fog on - example

- **Numpad 0**: Record the camera's position and direction to a file for animation.
- **R**: Reload camera animation from file.
- **P**: Toggle flashlight on and off.

To interact with the system, the user simply runs the program, and the camera can be moved using the keyboard and mouse.

Chapter 5

Conclusions and Further Developments

The project successfully implements a basic 3D engine with real-time lighting and shadows, offering an interactive scene with a movable camera and multiple light sources and shadows applied. Further developments for this project include:

- More light sources.
- Advanced lighting techniques, such as bloom or HDR.
- Improved shadow algorithms like cascaded shadow maps.
- More complex scene interactions and object manipulation.
- Extra key bindings for a better user experience.
- Objects animations and collision detections.
- Transforming the scene into a small game.

Chapter 6

References

The following resources were used in the development of this project:

- **Real-Time Rendering** by Tomas Akenine-Möller, Eric Haines, and Meredith. A comprehensive textbook covering graphics algorithms.
- **OpenGL Shading Language (GLSL)** documentation. Available at: `https://www.khronos.org/opengl/wiki/GLSL`. Last accessed: December 2024.
- **Introduction to Modern OpenGL** by Randi J. Rost. A comprehensive guide to modern OpenGL programming. Available at: `https://www.oreilly.com/library/view/learning-opengl/9780321897231/`. Last accessed: January 2025.
- **OpenGL SuperBible** by Graham Sellers, Richard S. Wright, and Nicholas Haemel. A detailed tutorial on OpenGL programming. Available at: `https://www.openglsuperbible.com/`. Last accessed: January 2025.
- **LearnOpenGL website**. A comprehensive online resource for learning OpenGL. Available at: `https://learnopengl.com/`. Last accessed: January 2025.
- **C**ourse and laboratories materials.