**Photorealistic presentation of 3D objects using OpenGL**

Graphic processing Laboratory

Student: Cornea horea – ionut

Group: 30433

Table of Contents

[1. Introduction 2](#_Toc186497553)

[Objectives 2](#_Toc186497554)

[2. Subject Specification 2](#_Toc186497555)

[3. Scenario 2](#_Toc186497556)

[Scene and Object Description 3](#_Toc186497557)

[FUNCTIONALITIES 3](#_Toc186497558)

[4. Implementation Details 4](#_Toc186497559)

[Functions and Special Algorithms 4](#_Toc186497560)

[Possible Solutions 4](#_Toc186497561)

[Chosen Approach 5](#_Toc186497562)

[Graphics model 5](#_Toc186497563)

[Data structures 7](#_Toc186497564)

[Class Hierarchy 9](#_Toc186497565)

[5. Graphical User Interface 11](#_Toc186497566)

[6. User Manual 12](#_Toc186497567)

[7. Conclusions and Further Developments 13](#_Toc186497568)

# Introduction

This project demonstrates advanced OpenGL rendering techniques. It includes dynamic features such as real-time shadow computation, rain particle simulation, and a skybox environment. Users can interact with the simulation using keybindings for camera controls and toggling visual effects.

## Objectives

The main goals of this project are:

* Implement a realistic environment using shaders and OpenGL techniques.
* Provide an interactive platform to explore 3D scenes with dynamic lighting and effects.

# Subject Specification

This project is a 3D graphical simulation developed using OpenGL. It demonstrates the implementation of advanced rendering techniques, such as:

* **Dynamic Lighting**: A directional light that simulates sunlight.
* **Punctiform Lighting**: Multiple point lights add localized illumination.
* **Shadow Mapping**: An algorithm that calculates shadows dynamically based on the scene’s directional light source.
* **Rain Particle System**: Simulates weather effects by rendering thousands of rain/snow particles with realistic movement.
* **Skybox Rendering**: Uses a cubemap texture to create a visually immersive background.
* **Fog Rendering**: Simulates atmospheric depth by blending object colors with fog color based on distance, adding realism and enhancing ambiance.

# Scenario

## Scene and Object Description

The scene presents a medieval environment featuring a large stone castle, a small village with wooden houses and several decorations, and a nearby farming area with crops and animals. The landscape includes hills, a sandy shore, and a water body, surrounded by trees for a natural touch.

In order to achive this scene, multiple objects categories have been used:

* Ground: The ground is built from more object with different materials and textures:
* GRASS – which is the main part of the ground representing the grass
* SNOW – the snow in the mountain’s peak
* WATER – a flat plane object representing the lake
* FLOWERS – part of the ground that has flowers
* SAND – delimiting the lake from the grass object with sand
* IARBA\_SNOW – the same object IARBA but with a different material for the snowing scene
* FLORI\_SNOW – the same object FLORI but with a different material for the snowing scene
* TREE : There are multiple trees in the scene rendered from a single object called TREES
* DECORATION: In this category there are multiple objects that represent all the props and decorations in the scene. Some of them are:
* HOUESE, CORNFIELD, PROPELLER, WELL, HAY, LAMP, MARKET, WINDMILL
* SUN: The object that is needed for the representation of the orbiting sun which is the source for the directional light.
* RAIN: Is a simple quad object that is used for the rain and the snow particles.

## FUNCTIONALITIES

* **Directional Lighting**: The scene uses directional light to simulate sunlight. This lighting is uniform and provides realistic illumination across the entire environment. It interacts with the shadow-mapping algorithm to calculate dynamic shadows based on the position and orientation of the sun(directional light).
* **Shadows**:  Shadows are computed dynamically using a shadow mapping technique, ensuring that objects block light realistically, adding depth and visual authenticity to the environment.
* **Punctiform Lighting**: Punctiform (point) lighting represents localized light sources, (in this case, lanterns) . It uses an attenuation model to calculate the light intensity based on the distance from the light source to the affected objects, enhancing the realism of localized illumination.
* **Night Mode**: This functionality alters the scene’s lighting to simulate nighttime conditions. The directional light is turned off, cooler color tones are applied to mimic the night’s ambiental light and punctiform lights are turned on.
* **Fog Mode**: A volumetric effect is added to simulate atmospheric scattering. Fog mode introduces a gradient of reduced visibility with increasing distance, providing a realistic depth effect, especially in large, open environments.
* **Wireframe Mode**: A debugging feature that renders objects in a wireframe format, showing only the edges of the 3D models. This mode is used to inspect the structure and geometry of the models.
* **Primitive View**: A debugging feature that renders objects’s vertices.

* **Snow and Rain Effects**: The scene includes particle systems for weather effects. Snow particles are rendered as slowly falling objects, creating a calm, wintry effect, while rain particles fall rapidly with transparency to simulate raindrops. Both systems use physics-based updates for realistic movement and interactions. For the rain effect, a lighning effect is also visible.
* **Animation**: The scene features two animations: an automatic camera movement through a predefined path, providing a cinematic view of the environment, and spinning windmill propellers, implemented with continuous rotation controlled by a transformation matrix. These animations add dynamism and enhance the scene’s visual appeal.

# Implementation Details

This section provides an in-depth analysis of the implementation techniques, algorithms, data structures, and class hierarchy used in the project. Each subsection is detailed to cover the underlying logic, design choices, and their practical application in the scene.

## Functions and Special Algorithms

The project employs advanced algorithms and custom functions for efficient rendering, lighting, and environmental effects.

## Possible Solutions

1. **Shadow Calculation**:

* **Shadow Volumes**:

Shadow volumes rely on constructing 3D volumes that represent the shadowed regions of objects. These volumes are projected from the object’s edges based on the light source’s position. This approach provides precise and sharp shadows but comes with significant computational overhead, especially in scenes with a high number of dynamic objects.

* **Shadow Mapping**:

Shadow mapping is a texture-based approach that stores depth information from the light’s perspective in a depth map. During rendering, this depth map is compared against the scene’s fragment depth values to determine whether a fragment is in shadow. This method is less resource-intensive and works efficiently for dynamic scenes with multiple moving objects and lights.

2. **Particle Systems for Rain and Snow**:

* **GPU-Based Particles**:

A GPU-based particle system calculates the movement and behavior of particles directly on the GPU using compute shaders or vertex shader transformations. This method allows for rendering a vast number of particles simultaneously and is optimal for high-performance applications. However, it requires complex shader programming and advanced GPU support.

* **CPU-Based Particles**:

In a CPU-based system, the particle behavior is calculated on the CPU, and the updated data is passed to the GPU for rendering. This approach is simpler to implement and debug, but it can be a bottleneck for large-scale particle simulations due to the data transfer between the CPU and GPU.

3. **Fog Effect**:

* **Volumetric Fog**:

Simulates light scattering within a 3D volume, providing realistic atmospheric effects. However, it is computationally expensive and challenging to implement for real-time rendering.

* **Linear Fog**:

A simple approach that reduces the visibility of objects based on their distance from the camera. It is computationally efficient and visually effective, especially in outdoor scenes.

## Chosen Approach

* **Shadow Mapping** was selected for its flexibility and balance between computational efficiency and visual quality. It supports directional and point light sources and adapts well to dynamic environments.

**Pseudocode for computeShadow function in main fragment shader:**

1. Transform fragment position from light space to normalized device coordinates:

- projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w

- Scale to [0, 1] range: projCoords = projCoords \* 0.5 + 0.5

2. Check if fragment is outside the shadow map:

- If projCoords.z > 1.0, return 0.0 (lit).

3. Calculate the depth and bias:

- currentDepth = projCoords.z

- bias = max(0.002 \* (1.0 - dot(normal, lightDir)), 0.005)

4. Perform Percentage Closer Filtering (PCF):

- Initialize shadow to 0.0

- For each offset in a 3x3 grid:

- Get closestDepth from shadow map using texture coordinates.

- Increment shadow if currentDepth > closestDepth + bias.

5. Normalize shadow:

- shadow /= 9.0 (average over 3x3 samples).

6. Return shadow.

* **CPU-Based Particles** were used for snow and rain effects due to their simplicity and ease of debugging. The performance tradeoff was acceptable, given the manageable number of particles in the scene.

**Pseudocode for particle generation function:**

1. Initialize Particles:

- For each particle:

- Set random initial position within a defined area.

- Assign a random lifespan.

- Determine velocity based on whether the particle represents snow or rain.

2. Update Particles Each Frame:

- For each particle:

- Decrease lifespan by delta time.

- If lifespan > 0:

- Update position based on velocity.

- For snow, apply additional horizontal drift.

- If lifespan <= 0:

- Respawn particle above the camera with a new position and lifespan.

3. Render Particles:

- For each visible particle:

- Set position, size, and color (gray for rain, white for snow).

- Render particle model to the screen.

4. Optimize Performance:

- Cull particles that are too far from the camera to minimize rendering overhead.

* **Linear Fog** was chosen as it integrates seamlessly with the scene and provides a convincing depth effect without imposing a heavy computational load.

**Pseudocode for computeFog function in main fragment shader:**

1. Set fog density based on conditions:

- If rain is enabled:

- fogDensity = 0.03 (night) or 0.01 (day)

- Otherwise:

- fogDensity = 0.04 (night) or 0.02 (day)

2. Calculate distance to fragment:

- dist = length(fragPosEye.xyz)

3. Compute fog factor using exponential decay:

- fogFactor = exp(-pow(dist \* fogDensity, 2.0))

4. Clamp fog factor to range [0, 1]:

- fogFactor = clamp(fogFactor, 0.0, 1.0)

5. Return fogFactor.

## Graphics model

The graphics model adheres to the modern OpenGL rendering pipeline and incorporates techniques like shadow mapping, Phong shading, and particle systems.

1. **Transformation Pipeline**:

* **Model Transformation**:

Converts object vertices from local object space to world space using the model matrix.

* **View Transformation**:

Applies the camera’s position and orientation to transform world-space coordinates into view-space coordinates.

* **Projection Transformation**:

Maps 3D coordinates into clip space for rendering on a 2D screen.

The final transformation for each vertex is calculated as:

2. **Lighting Model**:

The scene uses the Phong shading model for realistic lighting:

* **Ambient Lighting**: Adds a base level of light to all surfaces to simulate indirect light.
* **Diffuse Lighting**: Depends on the angle between the surface normal and the light direction, providing realistic shading based on light intensity.
* **Specular Lighting**: Produces highlights based on the viewer’s angle relative to the light source.

3. **Shadow Mapping**:

* The light’s perspective is used to render a depth map, which stores the depth of the nearest surface relative to the light source.
* During the main rendering pass, the fragment depth is compared with the depth map to determine whether the fragment is in shadow.

4. **Weather Effects**:

* **Rain**: Each particle’s position and lifespan are updated every frame, simulating falling raindrops.
* **Snow**: Slow-moving particles simulate snowfall, creating a calm atmosphere. The particle system also has a driftVariation which makes the particles behave like there is actual wind blowing.

5. **Fog Calculation**:

* The fog factor is calculated based on the distance between the camera and each fragment:
* The final color is blended between the object color and the fog color using the fog factor.

## Data structures

Efficient and organized data structures are critical for managing the complexity of a 3D graphics application. In this project, several data structures are implemented to handle particles, meshes, textures, lighting, and camera operations. Each data structure is designed to optimize performance, maintain scalability, and simplify interactions between different components of the scene.

1. **Particles**

The particle system uses a vector-based structure to manage individual particles. Each particle is represented by a custom structure that stores its position, velocity, and lifespan. This design allows efficient updates and rendering of particles for effects like rain and snow.

struct Particle {

    glm::vec3 Position;  // Current position of the particle in world space

    glm::vec3 Velocity;  // Velocity vector determining movement

    float Life;          // Remaining lifespan; particles are respawned when Life <= 0

};

Particles are stored in a std::vector, which provides dynamic resizing and efficient iteration:

std::vector<Particle> particles;

The use of a vector ensures that particles can be easily added, updated, or removed as needed, which is crucial for real-time effects.

1. **Mesh Representation**

The scene’s objects are represented as meshes, which are collections of vertices, indices, and textures. Each mesh is defined by the following structure:

struct Vertex {

    glm::vec3 Position;  // 3D coordinates of the vertex

    glm::vec3 Normal;    // Normal vector for lighting calculations

    glm::vec2 TexCoords; // Texture coordinates for UV mapping

};

The vertices are organized in a vector, and their connectivity is defined by indices:

std::vector<Vertex> vertices;      // List of all vertices in the mesh

std::vector<unsigned int> indices; // Indices defining how vertices form triangles

std::vector<Texture> textures;     // Textures applied to the mesh

The indices reduce redundancy by reusing shared vertices between triangles, improving memory efficiency.

1. **Lighting System**

Lighting in the scene involves both directional and point lights. Each type of light is represented by a structure tailored to its specific properties.

• **Directional Light**:

struct DirectionalLight {

    glm::vec3 Direction;  // Direction vector of the light

    glm::vec3 Color;      // RGB color of the light

};

Directional lights simulate sunlight, providing uniform illumination across the scene. The Direction vector determines where the light originates.

• **Point Light**:

struct PointLight {

    glm::vec3 Position;   // Position of the point light in world space

    glm::vec3 Color;      // RGB color of the light

    float Attenuation;    // Light intensity decay over distance

};

Point lights are localized light sources with intensity that decreases based on the distance to the object. The Attenuation factor models this decay.

A vector of lights is maintained to support multiple light sources:

std::vector<PointLight> pointLights;

DirectionalLight directionalLight;

1. **Camera**

The camera is responsible for user navigation and perspective control. Its state is stored in a structure that includes position, orientation, and movement parameters.

struct Camera {

    glm::vec3 Position;   // Current position of the camera in world space

    glm::vec3 Front;      // Forward direction the camera is facing

    glm::vec3 Up;         // Upward direction for the camera

    float Yaw;            // Horizontal rotation angle

    float Pitch;          // Vertical rotation angle

};

The camera’s view matrix is calculated based on its position and direction, ensuring accurate transformations for rendering.

1. **Skybox**

The skybox is represented as a cubemap texture with six faces. The structure below defines the cubemap and its associated data:

struct Skybox {

    GLuint TextureID;      // OpenGL texture ID for the cubemap

    std::vector<std::string> Faces; // Filepaths for each cubemap face (right, left, top, bottom, front, back)

};

The TextureID is used during rendering to sample the cubemap, and the Faces vector ensures each face of the cubemap is loaded correctly.

1. **Animation Paths**

For automated camera movement, paths are represented as a sequence of waypoints, each with a position and a time step:

struct Waypoint {

    glm::vec3 Position; // Position of the waypoint

    float Time;         // Time at which the camera reaches this point

};

The camera interpolates between waypoints to create smooth transitions during automated animations.

**Summary of Data Structures**

These data structures collectively handle all aspects of the scene, from rendering geometry to controlling environmental effects. The modular design ensures that each system (e.g., particles, lighting, or camera) can be developed and maintained independently, while their integration results in a cohesive and realistic 3D environment.

## Class Hierarchy

The project follows an object-oriented design, with classes organized to handle specific responsibilities. This modular approach ensures maintainability, scalability, and clear separation of concerns. Below is a detailed description of the class hierarchy, including their roles and relationships.

1. **Shader Class**

**Purpose**: Manages the lifecycle of GLSL shaders, including compilation, linking, and usage during rendering.

**Key Responsibilities**:

* + Load vertex and fragment shader source code from files.
  + Compile and link shaders into a program.
  + Set uniform variables for transformations, lighting, and material properties.

1. **Camera Class**

**Purpose**: Provides a dynamic view into the 3D scene by managing the camera’s position, orientation, and movement.

**Key Responsibilities**:

* + Maintain the camera’s position, direction, and orientation.
  + Compute the view matrix for rendering transformations.
  + Respond to user input to enable free movement and rotation.

1. **Model3D Class**

**Purpose**: Represents and renders complex 3D models, typically composed of multiple meshes.

**Key Responsibilities**:

* + Load 3D models from OBJ files using the tiny\_obj\_loader library.
  + Store and manage meshes, materials, and textures.
  + Render the model by iterating through its meshes.

1. **Mesh Class**

**Purpose**: Represents a single 3D mesh, including its vertex data, indices, and associated textures.

**Key Responsibilities**:

* + Store vertex, index, and texture data.
  + Manage buffers and attributes for rendering.
  + Render the mesh with a specified shader.

1. **ParticleSystem Class**

**Purpose**: Simulates and renders particle-based effects like rain and snow.

**Key Responsibilities**:

* + Manage a collection of particles with properties like position, velocity, and lifespan.
  + Update particle positions each frame based on physics calculations.
  + Render particles using a quad geometry.

1. **LightManager Class**

**Purpose**: Centralizes the management of lighting in the scene, including both directional and point lights.

**Key Responsibilities**:

* + Store and update properties of all light sources.
  + Pass light parameters to shaders during rendering.

1. **Skybox Class**

**Purpose**: Renders the background environment using a cubemap texture.

**Key Responsibilities**:

* + Load and manage cubemap textures.
  + Render the skybox as a cube surrounding the scene.

1. **Fog Class**

**Purpose**: Implements and manages fog effects in the scene.

**Key Responsibilities**:

* + Store fog parameters such as start distance, end distance, and color.
  + Pass fog properties to shaders for rendering.

1. **Main Class**

**Purpose**: The **Main Class** serves as the central entry point and coordinator for the application. It initializes the core components, sets up the rendering loop, and handles user interactions, ensuring a smooth execution of the 3D scene.

**Key Responsibilities**:

* Application Initialization: Setting up the OpenGL context, creating the window, and initializing necessary libraries like GLEW or GLFW.
* Resource Management: Loading shaders, models, textures, and setting up buffers for rendering.
* Event Handling: Processing user input such as keyboard and mouse interactions.
* Rendering Loop: Managing the frame-by-frame rendering process, including updates to animations, particle effects, and lighting.
* Cleanup: Releasing resources and gracefully terminating the application.

# Graphical User Interface

The graphical user interface (GUI) of the application is minimalistic, focusing on interactive elements controlled via keyboard and mouse inputs. The program is designed to provide an immersive experience with intuitive controls, allowing users to navigate the 3D scene, explore features like weather effects and lighting, and interact with dynamic elements.

# User Manual

The application allows users to explore the 3D scene and interact with various features using the following controls:

**Camera Controls**

* + **W/A/S/D**: Move the camera forward, left, backward, or right for navigation within the scene.

**Lighting Controls**

* + **Left Arrow (←)**: Decrease the light angle, adjusting the sun’s position and shadow direction.
  + **Right Arrow (→)**: Increase the light angle, moving the sun and modifying shadow placement.

**Feature Toggles**

* + **1**: Enable **Night Mode**, dimming the scene and applying cooler tones for nighttime simulation.
  + **2**: Enable **Fog Mode**, adding atmospheric fog to reduce visibility at greater distances.
  + **3**: Enable **Animation**, automatic camera movement following some waypoints
  + **4**: Enable **Wireframe Mode**, displaying the geometry of objects in a wireframe format for debugging or analysis.
  + **5**: Enable **Primitive View**, displaying the vertices of objects in a point format for debugging or analysis.
  + **6**: Enable **Snow**, simulating a snowfall effect using a particle system.
  + **7**: Enable **Rain**, adding a rain effect with realistic falling droplets.

**Debugging and Information**

* + **0**: Retrieve the current camera coordinates, useful for debugging or setting specific viewpoints.

# Conclusions and Further Developments

**Conclusions**

The project demonstrates the successful implementation of a 3D scene with advanced rendering techniques and interactive functionalities. Key achievements include:

1. **Realistic Lighting and Shadows**:

• Directional and point lights simulate natural and localized lighting effectively, with dynamic shadows calculated using shadow mapping.

1. **Interactive Features**:

• The inclusion of night mode, fog effects, wireframe mode, and weather systems (rain and snow) enhances user engagement and provides an immersive experience.

1. **Animation**:

• The automated camera movement and windmill propeller rotation add dynamism, showcasing how animations integrate seamlessly with the environment.

1. **Weather Effects**:

• The rain and snow particle systems create visually appealing weather simulations, enriching the scene’s realism.

1. **Robust User Interaction**:

• The implementation of intuitive controls for camera navigation and feature toggling ensures accessibility and ease of use.

Overall, the project successfully combines advanced OpenGL techniques, efficient resource management, and interactive elements, resulting in a cohesive and visually engaging 3D application.

**Further Developments**

While the project achieves its primary goals, there are several opportunities for improvement and extension:

1. **Dynamic Weather Transitions**:

• Implement smooth transitions between rain, snow, and clear skies, simulating real-time weather changes.

1. **Post-Processing Effects**:

• Add effects like bloom, motion blur, and HDR to enhance visual quality and realism.

1. **Enhanced Fog**:

• Upgrade from linear fog to volumetric fog for more realistic atmospheric scattering.

1. **Advanced Shadows**:

• Implement soft shadows using percentage-closer filtering (PCF) or cascaded shadow maps for improved shadow quality and depth.

1. **Expanded Interaction**:

• Introduce interactive elements in the scene, such as opening doors, triggering events, or manipulating objects.

1. **Performance Optimization**:

• Optimize the particle systems by transitioning to GPU-based particle simulations for improved performance, especially for high particle counts.

1. **Support for Additional Projections**:

• Integrate additional projection modes, such as orthographic projection, to offer alternative viewing experiences.

1. **Real-Time Day/Night Cycle**:

• Add a dynamic day-night cycle with gradual changes in lighting, shadows, and skybox textures.

# References

1. Particle Systems: <https://learnopengl.com/In-Practice/2D-Game/Particles>
2. Shadow Mapping: <https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping>
   * + Laboratory Guide 9
3. Cubemaps (Skybox) rendering: <https://learnopengl.com/Advanced-OpenGL/Cubemaps>
4. Blender Tutorials:  
   <https://www.youtube.com/playlist?list=PLrgcDEgRZ_kndoWmRkAK4Y7ToJdOf-OSM>
5. Troubleshooting: <https://chatgpt.com>
6. 3D objects: <https://www.cgtrader.com>