UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Graphical Processing OpenGl Project

Moruţan Maria

Group 30433

20.01.2026

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Table of Contents

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

1. Subject specification

The project focuses on the development of a real-time 3D scene using OpenGL, integrating multiple aspects of modern graphics programming, including:

- Scene visualization with **camera movement, object transformation, and animation**.
- Scene and objects aspect modelling in **Blender**.
- Implementation of **multiple light sources**: directional light and spotlight.
- Application of **textures and materials** with proper quality mapping.
- Viewing **solid** versus **wireframe** objects, **polygonal** versus **smooth** surfaces
- Shadow computation for enhanced realism.
- User interaction via **keyboard and mouse**.

The goal is to create a **photo-realistic, interactive scene** where the user can explore the environment and interact with objects, observing lighting effects, shadows, and animations.

2. Scenario

2.1 Scene and objects description

The scene contains the following objects:

- **Cottage:** The central building of the scene, placed at the origin `(0,0,0)` with initial rotation and scaling exported from Blender.
- **Ground:** A plane textured with a high-resolution 1024×1024 map, scaled to cover the entire play area.
- **Dog (catel):** User-transformable object with keyboard-controlled Rotation ®.
- **Seagull(gull):** Animated object flying in a circular path with orientation always facing the cottage.
- **Water (apa), Cat (normalcat), Trees (padure1), Skybox (sky):** Supporting elements for realism.

The scene is **3D interactive**, designed to provide **photo-realism with shadows, fog, rain, and lighting effects**.

2.2 Functionalities

- **Camera movement:** Controlled with `W`, `A`, `S`, `D` keys, constrained to ground height. Mouse controls pitch and yaw.
- **Object transformations:** Dog and Cottage can be rotated with keyboard input.
- **Animation:** Seagull flies in a circular trajectory around the cottage.
- **Atmospheric Effects:** Toggleable fog and a particle-based rain system.
- **Lighting:**
  - Directional light simulating sun/moon with shadows.
  - Spotlight attached to camera, controllable via orientation.

Rendering options:
- Toggle solid/wireframe mode (`1`/`2`).
- Flat/smooth shading toggle (`3`/`4`).
- Fog toggle (`5`).

- Rain toggle (`6`).
- Shadow map visualization (`M`).

3. Implementation Details

3.1. Functions and Special Algorithms

3.1.1. Camera System and Navigation

The camera implementation uses a **Synthetic Camera Model** based on the Gram-Schmidt orthonormalization process.

3.1.1.1. Movement Logic

The camera movement is decoupled from the frame rate by using a `cameraSpeed` factor. The position is updated based on the "Front" and "Right" vectors:

- **Forward/Backward:** Position = Position +- (Front * speed)
- **Left/Right:** Position = Position +- (normalize(Front * Up* speed)$

3.1.1.2. View Matrix Animation (Mouse Look)

To achieve smooth rotation, we use Euler Angles: Yaw (horizontal) and Pitch (vertical). The direction vector is calculated using spherical coordinates converted to Cartesian:

$$x = \cos(yaw) * \cos(pitch)$$

$$y = \sin(pitch)$$

$$z = \sin(yaw) *\backslash \cos(pitch)$$

The final View Matrix is generated using the **glm::lookAt** function, which creates a coordinate system where the camera is the origin.

3.1.2. Lighting and Shading Model

The scene utilizes a hybrid lighting model combining **Global Directional Lighting** and a **Localized Spotlight**.

3.1.2.1. Directional Light (The Sun)

The sun is implemented as a light source at "infinity," meaning all light rays are parallel. The intensity is calculated using the Lambertian Reflection model:

Diffuse = max(L * N, 0) * LightColor

Where **L** is the light direction vector and **N** is the surface normal. There is also a Light Rotation matrix implemented that allows the sun to orbit the scene, calculated using:

lightRotation = glm::rotate(glm::mat4(1.0f), glm::radians(lightAngle), glm::vec3(0, 1, 0));

3.1.2.2. Dynamic Flashlight (Spotlight)

The spotlight is attached to the camera's position and direction. It uses a **Dual-Cone** approach to create soft edges:

1. **Inner Cutoff:** Full intensity.
2. **Outer Cutoff:** Light fades to zero.
3. Intensity calculation: I = clamp((theta - Outer) / (Inner - Outer), 0.0, 1.0)

   Where **theta** is the cosine of the angle between the light direction and the vector to the fragment.

3.1.3. Animation and Special Algorithms

3.1.3.1. Procedural Animations

The scene includes two distinct types of animation:

- **The Seagull (Gull):** Uses a circular trajectory calculated using `sin(t)` and `cos(t)`. The model is automatically rotated to face the direction of movement using `atan2(dx, dz)`.
- **The Rain System:** A particle system where 2,000 vertices are updated every frame. When a "drop" reaches Y < -1.5, its position is reset to Y = 40.0, creating a continuous loop of falling rain without the overhead of instantiating new objects.

3.1.3.2. Shadow Mapping Algorithm

Shadows are implemented via **Depth Buffering**. During the first pass, the scene is rendered from the light's POV using an **Orthographic Projection**.

- Shadow Bias: To prevent "Shadow Acne," a variable bias is applied based on the slope of the surface:

  float bias = max(0.05 * (1.0 - dot(normal, lightDir)), 0.005);

- **Over-sampling:** Fragments outside the light's frustum are set to "not in shadow" to prevent artifacts at the edges of the map.

3.1.4. Graphical User Interface Implementation

While the project lacks a traditional 2D GUI, the interface is handled through **GLFW Callbacks**.

- **State Management:** A global `RenderMode` enum and several booleans (`fogEnabled`, `rainEnabled`) act as the "model" of the application state.
- **Cursor Control:** The logic for the cursor: `GLFW_CURSOR_DISABLED` is used for immersive walking, while `GLFW_CURSOR_NORMAL` allows the user to interact with the window's close and minimize buttons. The immersive walking functionality can be accessed by pressing the key `ESC`, while the interaction with the window's close and minimize buttons can be achieved by pressing the `TAB` key. The application exits upon pressing the `X` key.

3.2. Possible Solutions

In developing the 3D Forest Cottage, several architectural paths were considered for handling the core graphics challenges.

A. Shadow Mapping vs. Ray Tracing

- **Ray Tracing:** Provides physically accurate shadows and reflections by tracing the path of light rays. However, it is computationally expensive and requires modern RTX hardware to run at interactive frame rates.
- **Shadow Mapping (Chosen):** A technique where the scene is rendered from the light's point of view to a "Depth Map." It is highly efficient for real-time applications and works on almost all hardware. I chose this because it allows for a high-poly cottage and 2,000 rain particles while maintaining 60+ FPS.

B. Forward Rendering vs. Deferred Shading

- **Deferred Shading:** Efficient for scenes with hundreds of light sources, as it separates geometry processing from lighting. However, it is complex to implement with transparency (like our rain and water) and uses more video memory.
- **Forward Rendering (Chosen):** The standard OpenGL approach where each object is drawn and shaded one by one. Since our scene relies on a few high-quality lights (Sun + Flashlight), Forward Rendering was the most effective solution for maintaining high-quality anti-aliasing and handling transparent textures.

C. Shading Models: Gouraud vs. Phong vs. Blinn-Phong

- **Gouraud Shading:** Calculates lighting at the vertices and interpolates across the face. While fast, it creates "blocky" highlights on the cottage walls.
- **Blinn-Phong (Chosen):** Calculates lighting at every pixel (fragment). We chose this because it produces smooth, realistic specular highlights on the water surface and the cottage roof, especially when the spotlight is pointed directly at them.

3.3. Motivation for the Chosen Approach

The final architecture was chosen to balance **visual fidelity** with **performance**.

1. **Uniform-Driven State Management:** By using `glUniform`, we allow the user to toggle the fog and rain without re-loading shaders. This makes the "Scenario" (Section 3) interactive and seamless.
2. **Particle System Efficiency:** Instead of creating 2,000 "Object" instances for rain (which would crash the CPU), I used a single model and iterated through an array of `glm::vec3` positions. This moves the heavy lifting to the GPU's transformation pipeline.
3. **Shadow Bias:** To solve the common issue of "Shadow Acne" (jagged lines on the ground), we implemented a variable bias based on the angle of the light.
    - *Algorithm:* bias = max(0.05 * (1.0 - * (N, L)), 0.005)
    - This ensures that shadows look crisp on the cottage but don't "flicker" on the terrain.

3.4. Graphics Model

The project utilizes the **Blinn-Phong Lighting Model** integrated with an **Exponential Squared Fog** model and **Shadow Mapping**.

Lighting Model

The final color of a pixel is determined by the sum of a directional light and a camera-attached spotlight:

Color_final = Ambient + (1.0 - Shadow) * (Diffuse + Specular) + Spotlight

- **Shadow:** Calculated using a depth-comparison algorithm. If the current fragment's depth is greater than the value stored in the shadow map, it is multiplied by a shadow factor.
- Spotlight: Implemented with a "Smooth Step" transition to prevent hard edges.

Fog Model

To simulate atmospheric depth, we implemented Exponential Squared Fog. This is more realistic than linear fog as it thickens faster as distance increases.

The final fragment color is mixed with the fog color based on this factor:

mix(fogColor, finalColor, F).

3.5. Data Structures

The application relies on specific data structures to manage the complexity of 3D data and state:

- **`std::vector<Mesh>`**: Contained within the `Model3D` class. Each mesh stores a vector of `Vertex` structures (containing position, normal, and texture coordinates) and a vector of `unsigned int` for indices.
- **`glm::mat4`**: High-performance 4x4 floating-point matrices used for transformations. The project utilizes a "Model-View-Projection" (MVP) stack.
- **`GLuint` (Handles)**: Used to store references to GPU-side resources like the Shadow Map FBO, Texture Units, and Shader Programs.
- **`bool pressedKeys[1024]`**: A boolean map that tracks the state of every key on the keyboard simultaneously. This allows for fluid "diagonal" movement (pressing W and A at the same time).

3.6. Class Hierarchy

The project follows a modular, Object-Oriented Programming (OOP) approach to keep the OpenGL state manageable.

1. `gps::Camera`

- **Responsibility:** Manages the View Matrix.
- **Key Attributes:** `cameraPosition`, `cameraFrontDirection`, `cameraRightDirection`.
- **Key Methods:** `getViewMatrix()`, `move()`, `rotate()`.

2. `gps::Model3D`

- **Responsibility:** Interface for the Assimp library. It loads 3D meshes and their associated materials (textures).
- **Interaction:** Contains a list of `Mesh` objects. It is called during both the Depth Pass and the Final Pass.

3. `gps::Shader`

- **Responsibility:** Compiles and links Vertex and Fragment shaders.
- **Key Methods:** `loadShader()`, `useShaderProgram()`. It provides the `shaderProgram` ID used by `glUniform` calls to update lighting variables.

4. `gps::Mesh`

- **Responsibility:** The lowest level of geometry. It manages VAOs (Vertex Array Objects) and VBOs (Vertex Buffer Objects).
- **Interaction:** Communicates directly with the GPU using `glDrawElements`.

4. Graphical User Interface Presentation / User Manual

The application utilizes a **minimalist, control-based GUI** integrated directly into the 3D viewport. Interaction is handled via a combination of mouse-look and keyboard state polling.

4.1. Window Controls and Modes

The application starts in **Immersive Mode** (Cursor Disabled).

- **Window Management:** The window is fully resizable. Upon maximizing, the `windowResizeCallback` recalculates the aspect ratio to prevent geometry distortion.
- **Control Toggling:**
  - **TAB:** Re-enters Immersive Mode (locks cursor to center for navigation).
  - **ESC:** Switches to **Window Mode**. The cursor becomes visible, allowing the user to minimize, move, or close the window via the standard 'X' button.

4.2. Keyboard Command Reference

The user can manipulate the environment using the following key bindings:

Category	Key	Action
Navigation	**W, A, S, D**	Forward, Left, Backward, Right (Camera relative)
Rotation	**Q, E, R**	Rotate the cottage object on the Y-axis, Rotate the dog object on the Y-axis
Lighting	**J, L**	Rotate the Sun (Directional Light) around the scene

Rendering	**1, 2**	Switch between **Solid** and **Wireframe** modes
Shading	**3, 4**	Toggle **Smooth Shading** vs. **Flat Shading**
Atmosphere	**5, 6**	Toggle **Fog, Rain** effect
System	**X**	Safely terminate the application

5. Conclusions and Further Developments

5.1. Conclusions

The project successfully demonstrates the core principles of the **Modern OpenGL Programmable Pipeline**. By moving away from the fixed-function pipeline, I achieved:

- **Dynamic Lighting:** Real-time calculation of spotlight attenuation and directional diffuse/specular components.
- **Atmospheric Realism:** The integration of exponential fog and a 2,000-particle rain system provides a cohesive environment.
- **Performance Stability:** The use of **FBOs** for shadow mapping and efficient data transfer via Uniforms allows the scene to maintain high frame rates even with complex shadows.

5.2. Further Developments

While the current version meets all subject specifications, several enhancements could be implemented:

1. **Skeletal Animation:** Replacing the current procedural translation of the dog and gull with vertex-skinning animations for more natural movement.
2. **PCF (Percentage Closer Filtering):** Implementing a kernel-based shadow sampler to soften jagged edges in the shadow map.
3. **Skybox Integration:** Replacing the simple sky mesh with a Cubemap texture for a more realistic 360-degree background.
4. **Collision Detection:** Implementing an AABB (Axis-Aligned Bounding Box) system to prevent the camera from walking through the cottage walls.

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

6. References

During the development of this project, the following resources were fundamental for implementing the graphics algorithms and shader logic:

1. **De Vries, J.** (2023). *Learn OpenGL: Shadow Mapping*. Available at: learnopengl.com. **[Highlighted: Foundation for Pass 1 and Pass 2 rendering logic]**.
2. **Shreiner, D., et al.** (2013). *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3*. Addison-Wesley Professional. **[Highlighted: Source for Phong lighting and Normal Matrix mathematics]**.
3. **GLSL Specification.** (2020). *Standard Derivative Functions*. Available at: khronos.org**. [Highlighted: Crucial for implementing Flat Shading using dFdx and dFdy]**.
4. **Khronos Group.** (2021). *Frame Buffer Objects (FBO)*. Available at: [suspicious link removed]. **[Highlighted: Technical implementation of off-screen depth textures]**.