GP Project Documentation

Tanase Luisa Elena

January 16, 2025

Contents

1 Contents

This documentation provides an overview of the project, including subject specifications, scenarios, implementation details, user manual, graphical interface and further developments.

2 Subject Specification

This project is an OpenGL application that renders a scene representing a tranquil natural setting. The main elements of the scene include a house nestled in nature, surrounded by lush trees, a well, and various animals that bring life to the environment. The objective is to create a realistic and immersive graphical representation using OpenGL.

3 Scenario

3.1 Scene and Objects Description

The scene consists of a ground object simulating some grass, surrounded by a skybox with some houses which are surrounded by multiple trees. On the ground, there are multiple object loaded:

1. a house
2. a flashlight on the table of the house's terrace
3. a cat
4. 2 dogs
5. a duck
6. a well with mobile handle
7. 4 trees of different sizes
8. a human in front of the house
9. a flying bat
10. 2 light lamps
11. a polytree
12. there can be some flower objects generated with a keyboard command

3.2 Functionalities

The scene is arranged by using multiple translations, rotations and scalings of the objects rendered, in order to create a visually appealing environment. The application provides camera movement by continuously pressing some keyboards, as follows:

- W - move forward
- S- move backward
- A - move left
- D - move right

The application also provides the camera animation functionality. If the user presses the key Z once, the camera will begin its animation. It will go from its start position closer to the center of the scene and then a bit left, in front of the leftmost object rendered on the scene. Then it will go right up until the rightmost object to present them all smoothly and then it will go back to its initial position. The process can be repeated if the key Z is pressed again.

The project has 3 different light sources: a directional light, 2 point lights, one for each lamp on the scene and a spotlight.

There is also the posibility to view the scene in wireframe mode (by pressing key T) or solid (by pressing key Y).

All of the objects rendered have textures mapped on them. The detailed textures make them look more realistic and enhance the effect of shininess on the materials.

The project implements shadow mapping, rendering shadows for each object loaded in the scene, relative to the directional light which can change its angle by pressing the keys J and L.

The application also exemplifies animation of object components. The handle of the well can rotate around itself, independently of the well object it is attached to.

For object generation exemplification, if the user presses the key G, flower objects will appear randomly on a certain area and each of the generated flowers will dissapear after a certain amount of time.

For collision detection, the duck object is "wrapped" in a AABB bounding box, and the well which is next to it too, such that when translated on x-axis or z-axis, the duck will not go trough the well.

The application also illustrates the fog effect on the scene.

If the user presses the key R then the scene will have rain. The rain is implemented by drawing animated small vertical lines which fall. The raindrops are drawn using rain.vert and rain.frag shaders.

The human seems like it breaths, by continuously doing a vert small scale transformation only on the y-axis. The bat above rotates and the trees illustrate an ingenious animation mimicking the wind effect.

The pointlights are enabled by pressing the key P, and the spotlight is enabled by pressing the key O.

4 Implementation Details

4.1 Functions and Special Algorithms

4.1.1 Wind effect animation

The following code snippet simulates a wind effect by applying a periodic rotation to tree models in the scene. The sway angle is calculated as a sine function of time, modulated by the wind's frequency and strength, with an additional random phase to vary the effect per tree. The model matrix is updated with this rotation around the Z-axis, creating the appearance of swaying trees. If not in the depth pass, the code recalculates the normal matrix to ensure proper lighting effects. Finally, the tree model is drawn using the updated transformation.

Listing 1: Wind Effect Implementation

```
float currentTime = glfwGetTime();
float windStrength = 2.0f; // Maximum sway angle in degrees
float windFrequency = 0.5f; // Oscillation frequency
```

```
float randomPhase = 2.0f;    // Per-tree randomization
    factor

float swayAngle = windStrength * sin(windFrequency *
    currentTime + randomPhase);
model = glm::rotate(model, glm::radians(swayAngle), glm::
    vec3(0.0f, 0.0f, 1.0f)); // Rotate around Z-axis

glUniformMatrix4fv(glGetUniformLocation(shader.
    shaderProgram, "model"), 1, GL_FALSE, glm::value_ptr(
    model));

if (!depthPass) {
     normalMatrix = glm::mat3(glm::inverseTranspose(view *
         model));
     glUniformMatrix3fv(normalMatrixLoc, 1, GL_FALSE, glm::
         value_ptr(normalMatrix));
}

tree.Draw(shader);
```

4.1.2 Camera animation

This function, animateCamera, handles the animation of the camera along a predefined path based on elapsed time since the animation started. The camera moves through various directions (forward, left, right, backward) in distinct time intervals, creating a sequence of movements. The startTimeStamp variable ensures the animation starts only once, resetting after completion. The moveCameraEnabled flag activates or deactivates the animation loop. A consistent speed for movement is maintained using the speed variable. By employing conditions based on elapsed time, the function makes the camera's path to achieve smooth and timed movements for an enhanced user experience.

Listing 2: Camera Animation Logic

```
double startTimeStamp = 0.0f;
void animateCamera() {

        float speed = 0.03f;

        if (!moveCameraEnabled) {
                return;
```

```
        }

        if (startTimeStamp == 0.0f) {
                myCamera.cameraPosition = glm::vec3(0.0
                    f, 2.2f, 7.5f);
                startTimeStamp = glfwGetTime();
        }

        double currentTimeStamp = glfwGetTime();
        double elapsedTime = currentTimeStamp -
            startTimeStamp;

        if (elapsedTime < 0.5) {
                myCamera.move(gps::MOVE_FORWARD, speed)
                    ;
        }
        else if (elapsedTime < 2.0) {
                myCamera.move(gps::MOVE_LEFT, speed);
        }
        if (elapsedTime < 2.5) {
                myCamera.move(gps::MOVE_FORWARD, speed)
                    ;
        }
        else if (elapsedTime < 5.5) {
                myCamera.move(gps::MOVE_RIGHT, speed);
        }
        else if (elapsedTime < 7.5) {
                myCamera.move(gps::MOVE_BACKWARD, speed
                    );
        }
        else if (elapsedTime < 9.25) {
                myCamera.move(gps::MOVE_LEFT, speed);
        }
        else {
                moveCameraEnabled = false;
                startTimeStamp = 0.0f;
        }
}
```

4.1.3 Collision detection

The following code snippet implements Axis-Aligned Bounding Box (AABB) collision detection and updates the position of a duck model. The AABB struct defines a bounding box using minimum and maximum coordinates. The computeDuckAABB function computes the transformed AABB for a duck, applying the model matrix transformations to its original bounds. The checkCollision function determines if two AABBs overlap by comparing their respective minimum and maximum coordinates in the X and Z axes.

The updateDuckPosition function calculates the duck's new position based on its current position and movement vector. It then computes the duck's AABB at the new position and checks for collisions with a predefined wellAABB. If no collision occurs, the duck's position is updated; otherwise, the movement is restricted, ensuring the duck does not intersect the well. This system provides efficient collision detection for interactive environments.

Listing 3: AABB Collision Detection and Duck Movement

```
struct AABB {
        glm::vec3 min;
        glm::vec3 max;
};

AABB wellAABB = { glm::vec3(2.9f, 0.0f, -2.4f), glm::
    vec3(3.7f, 0.0f, -1.6f) };

AABB computeDuckAABB(glm::mat4 model) {
        glm::vec3 duckMin(-0.1f, 0.0f, -0.1f);
        glm::vec3 duckMax(0.1f, 0.0f, 0.1f);

        glm::vec3 transformedMin = glm::vec3(model *
            glm::vec4(duckMin, 1.0f));
        glm::vec3 transformedMax = glm::vec3(model *
            glm::vec4(duckMax, 1.0f));

        return { transformedMin, transformedMax };
}

bool checkCollision(const AABB& a, const AABB& b) {
        return (a.max.x >= b.min.x && a.min.x <= b.max.
            x) &&
                (a.max.z >= b.min.z && a.min.z <= b.max
```

```
                    .z);
}

void updateDuckPosition(glm::vec3& position, const glm
    ::vec3& movement) {
        glm::vec3 newPosition = position + movement;

        glm::mat4 tempModelMatrix = glm::translate(glm
            ::mat4(1.0f), newPosition);

        AABB duckAABB = computeDuckAABB(tempModelMatrix
            );

        if (!checkCollision(duckAABB, wellAABB)) {
                position = newPosition;
        }
}
```

4.1.4 Object generation

The following code snippet dynamically generates and renders flower objects at random positions within a 3D scene. The Flower structure holds the creation time and random X and Z coordinates for each flower. When generateObj is true (meaning the user pressed the key G), a new flower is added to the activeFlowers list with randomized position values.

During rendering, the drawRandomObj function iterates through the active flowers. For each flower, it checks if less than 3 seconds have passed since its creation. If so, the flower is rendered by applying translation, scaling, and rotation transformations to its model matrix. Shader uniforms are updated accordingly, including the model matrix and the normal matrix for lighting calculations.

If a flower's lifetime exceeds 3 seconds, it is removed from the list. This approach ensures that the scene contains only active flowers while dynamically managing their creation and removal, maintaining visual variety and efficient memory usage.

Listing 4: Random Flower Object Generation and Rendering

```
struct Flower {
        double startTime;
        float randomX;
        float randomZ;
```

```
};
std::vector<Flower> activeFlowers;

void drawRandomObj(gps::Shader shader, bool depthPass)
    {
        if (generateObj) {
            Flower newFlower;
            newFlower.startTime = glfwGetTime();
            newFlower.randomX = -1.0f + static_cast
                <float>(rand()) / (RAND_MAX / 2.0f);
            newFlower.randomZ = 0.0f + static_cast<
                float>(rand()) / (RAND_MAX / 1.0f);

            activeFlowers.push_back(newFlower);
            generateObj = false;
        }

        for (size_t i = 0; i < activeFlowers.size(); )
            {
            double currentTime = glfwGetTime();
            double elapsedTime = currentTime -
                activeFlowers[i].startTime;

            if (elapsedTime < 3.0) {
                shader.useShaderProgram();
                glm::mat4 model = glm::mat4(1.0
                    f);

                model = glm::translate(model,
                    glm::vec3(activeFlowers[i].
                    randomX, -0.58f,
                    activeFlowers[i].randomZ));
                model = glm::scale(model, glm::
                    vec3(0.005f));
                model = glm::rotate(model, glm
                    ::radians(-90.0f), glm::vec3
                    (1.0f, 0.0f, 0.0f));

                glUniformMatrix4fv(
                    glGetUniformLocation(shader.
                    shaderProgram, "model"), 1,
```

```
                    GL_FALSE, glm::value_ptr(
                    model));

                if (!depthPass) {
                        normalMatrix = glm::
                            mat3(glm::
                            inverseTranspose(
                            view * model));
                        glUniformMatrix3fv(
                            normalMatrixLoc, 1,
                            GL_FALSE, glm::
                            value_ptr(
                            normalMatrix));
                }

                flower.Draw(shader);

                ++i;
            }
            else {
                activeFlowers.erase(
                    activeFlowers.begin() + i);
            }
        }
}
```

4.1.5 Rain

The following code snippet implements a dynamic rain simulation using OpenGL, with raindrops represented as lines. Each raindrop is a RainDrop struct containing its position, end position, speed, size, and creation time. The initializeRainBuffers function sets up OpenGL buffers to store vertex data and line indices for rendering.

Raindrops are generated in generateRainDrops, which assigns random positions and speeds to simulate natural variation. The drawRain function updates each raindrop's position over time, removing it if it falls below a certain height. Updated vertex data is uploaded to the GPU using glBuffer-SubData.

The updateRain function periodically generates new raindrops, ensuring a consistent rain effect. Together, these functions create a visually dynamic

rain simulation that efficiently manages resources and provides smooth rendering.

Listing 5: Rain Simulation with Dynamic Vertex Buffers

```
GLuint rainVBO, rainVAO, rainEBO;

struct RainDrop {
        glm::vec3 position;
        glm::vec3 endPosition;
        float speed;
        double startTime;
        float size;
};

std::vector<RainDrop> activeRainDrops;

void initializeRainBuffers() {
        glGenVertexArrays(1, &rainVAO);
        glGenBuffers(1, &rainVBO);
        glGenBuffers(1, &rainEBO);

        glBindVertexArray(rainVAO);

        glBindBuffer(GL_ARRAY_BUFFER, rainVBO);
        glBufferData(GL_ARRAY_BUFFER, sizeof(glm::vec3)
            * 2000, nullptr, GL_DYNAMIC_DRAW);

        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, rainEBO);
        std::vector<GLuint> indices(2000);
        for (size_t i = 0; i < 1000; ++i) {
                indices[i * 2] = i * 2;
                indices[i * 2 + 1] = i * 2 + 1;
        }
        glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.
           size() * sizeof(GLuint), indices.data(),
           GL_STATIC_DRAW);

        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,
            sizeof(glm::vec3), (void*)0);
        glEnableVertexAttribArray(0);
```

```
        glBindVertexArray(0);
        glBindBuffer(GL_ARRAY_BUFFER, 0);
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
}

void generateRainDrops(int count) {
        for (int i = 0; i < count; ++i) {
                RainDrop newRainDrop;
                newRainDrop.position = glm::vec3(
                        -1.0f + static_cast<float>(rand
                            ()) / (RAND_MAX / 2.0f),
                        1.0f,
                        -1.0f + static_cast<float>(rand
                            ()) / (RAND_MAX / 2.0f)
                );
                newRainDrop.speed = 0.5f + static_cast<
                    float>(rand()) / (RAND_MAX / 2.0f);
                newRainDrop.startTime = glfwGetTime();
                newRainDrop.size = 0.01f;

                newRainDrop.endPosition = newRainDrop.
                    position;
                newRainDrop.endPosition.y -= 0.05f;

                activeRainDrops.push_back(newRainDrop);
        }
}

void drawRain() {
        double currentTime = glfwGetTime();
        std::vector<glm::vec3> rainDropVertices;

        for (size_t i = 0; i < activeRainDrops.size();)
            {
                RainDrop& rainDrop = activeRainDrops[i
                    ];

                double elapsedTime = currentTime -
                    rainDrop.startTime;
                rainDrop.position.y -= rainDrop.speed *
                     static_cast<float>(elapsedTime);
```

```
                rainDrop.endPosition.y -= rainDrop.
                    speed * static_cast<float>(
                    elapsedTime);

                if (rainDrop.position.y < -0.35f) {
                        activeRainDrops.erase(
                            activeRainDrops.begin() + i)
                            ;
                } else {
                        rainDropVertices.push_back(
                            rainDrop.position);
                        rainDropVertices.push_back(
                            rainDrop.endPosition);
                        ++i;
                }
        }

        glBindBuffer(GL_ARRAY_BUFFER, rainVBO);
        glBufferSubData(GL_ARRAY_BUFFER, 0,
            rainDropVertices.size() * sizeof(glm::vec3),
             rainDropVertices.data());
        glBindBuffer(GL_ARRAY_BUFFER, 0);

        rainShader.useShaderProgram();
        glBindVertexArray(rainVAO);

        glDrawElements(GL_LINES, rainDropVertices.size
            (), GL_UNSIGNED_INT, 0);

        glBindVertexArray(0);
}

void updateRain() {
        static double lastSpawnTime = 0.0;
        double currentTime = glfwGetTime();

        if (currentTime - lastSpawnTime > 0.1) {
                generateRainDrops(10);
                lastSpawnTime = currentTime;
        }
}
```

4.1.6 The Motivation of the Chosen Approach

- The rain algorithm was chosen like this, i.e to render each raindrop as a line is better than rendering each raindrop as an object because that was too computationally expensive and it even made it difficult for the camera to move, making for the camera to be impossible to move smoothly.
- The object generation algorithm was implemented to make each flower disappear after 3 seconds, to create a more dynamic visual appearance on the scene and to be more efficient in terms of memory management.
- The collision detection algorithm is the classical one, which relies on "wrapping" objects in their AABB bounding boxes and check for collisions by verifying if the boxes intersect.

4.2 Graphics Model

To render a 3D .obj file in the OpenGL application, the .obj model along with its .mtl file and texture images were downloaded from a reliable online source. The .obj file contains 3D geometry (vertices, normals, texture coordinates), while the .mtl file specifies material properties and maps textures to the 3D object. The .mtl file was used to bind textures on the objects by mapping the corresponding pictures/images. Then, transformations were applied for positioning, scaling, and rotating each object in 3D space. The 3D scene is rendered by drawing each object with appropriate shaders for lighting, textures, and material effects to achieve a realistic appearance.

4.3 Data Structures

The data structures utilized in this project combine mathematical, graphical, and logical components to effectively handle the complexities of a 3D rendering engine. These are described below:

4.3.1 Matrices and Vectors

- `glm::mat4`: Utilized for 4x4 transformation matrices such as `model`, `view`, and `projection`, which manage object positioning, camera views, and perspective projection in 3D space.
- `glm::mat3`: Used for normal matrices, ensuring the proper transformation of normals for accurate lighting calculations.

- **`glm::vec3`**: Represents 3D vectors and points, used for spatial data like positions, directions (e.g., light or camera), and object transformations.

4.3.2 Structured Data

- **Structures (`struct`)**: Group related attributes into cohesive units for specialized tasks:
  - **`RainDrop`**: Contains attributes of a raindrop, including position, speed, size, and animation timing.
  - **`AABB (Axis-Aligned Bounding Box)`**: Encapsulates the minimum and maximum coordinates of an object for efficient collision detection.

4.3.3 Arrays and Buffers

- **Vertex Buffers (VBO, VAO, EBO)**: Handle rendering data such as vertices and indices, storing them in GPU memory for efficient drawing operations.
- **`std::vector`**: A dynamic container, such as `std::vector<RainDrop>`, used for tracking multiple objects (e.g., raindrops) in a flexible and efficient manner.

4.3.4 Objects and Models

- **Custom Classes**:
  - **`Model3D`**: Represents 3D objects, managing their geometry and textures for rendering.
  - **`Shader`**: Encapsulates shader programs, enabling advanced rendering techniques and real-time effects.

4.3.5 Flags and Controls

- **Boolean Flags (`bool`)**: Control the state of various functionalities, such as enabling camera movement (`moveCameraEnabled`) or object generation (`generateObj`).

4.3.6 Handles and Identifiers

- **OpenGL Identifiers (`GLuint`)**: Manage GPU resources such as textures, framebuffers, and shader programs (e.g., `depthMapTexture`, `shadowMapFBO`).

These data structures collectively provide a robust foundation for rendering dynamic and interactive 3D scenes, enabling features such as collision detection, real-time animation, and advanced lighting effects.

4.4 Class Hierarchy

4.4.1 Camera Class

The `Camera` class manages the camera's position, orientation, and movement in 3D space. Key components include:

- **Constructor**: Initializes the camera with position, target, and up direction. It calculates the front and right directions.
- **getViewMatrix()**: Returns the view matrix using `glm::lookAt()`.
- **move()**: Updates position based on directional input (forward, backward, left, right).
- **rotate()**: Adjusts orientation by modifying pitch (X-axis) and yaw (Y-axis).

The class uses the `glm` library for vector and matrix operations.

4.4.2 Mesh Class

The `Mesh` class handles 3D mesh data, including setup, storage, and rendering. Key components:

- **Constructor**: Initializes the mesh with vertex, index data, and textures, calling `setupMesh()`.
- **getBuffers()**: Returns buffers (VAO, VBO, EBO) for the mesh.
- **Draw()**: Renders the mesh with textures applied to the shader.
- **setupMesh()**: Initializes buffer objects and loads vertex/index data to the GPU.

The class relies on OpenGL functions for rendering and texture management.

4.4.3 Model3D Class

The `Model3D` class loads, parses, and draws 3D models from '.obj' files. Key components:

- **LoadModel()**: Loads models from file or specified base path.
- **Draw()**: Draws each mesh using the provided `Shader` program.
- **ReadOBJ()**: Parses vertex data, indices, and materials, creating mesh objects.
- **LoadTexture()**: Loads and stores textures.
- **ReadTextureFromFile()**: Loads textures into GPU memory, handling flipping and mipmaps.
- **Destructor**: Frees loaded textures and deletes OpenGL buffers.

The class uses `tinyobj` for parsing and OpenGL for rendering.

4.4.4 Shader Class

The `Shader` class manages the compilation and usage of OpenGL shaders. Key functions:

- **readShaderFile()**: Reads shader file contents into a string.
- **shaderCompileLog()**: Checks shader compilation status and prints errors.
- **shaderLinkLog()**: Checks program linking status and prints errors.
- **loadShader()**: Loads and compiles vertex and fragment shaders, linking the program.
- **useShaderProgram()**: Activates the shader program for rendering.

The class uses OpenGL functions for shader creation, compilation, and linking.

4.4.5 SkyBox Class

The `SkyBox` class creates and renders a skybox, simulating the surrounding environment. Key methods:

- **SkyBox()**: Constructor initializes an empty skybox.
- **Load()**: Loads cube map textures and initializes the skybox.
- **Draw()**: Renders the skybox using the shader and view/projection matrices.
- **LoadSkyBoxTextures()**: Loads texture faces into a cube map.
- **InitSkyBox()**: Initializes skybox geometry and VAO for rendering.
- **GetTextureId()**: Returns the cubemap texture ID.

The skybox is rendered by drawing a cube with textures, ensuring it appears behind other objects using `GL_LEQUAL` depth testing.

5 Graphical User Interface Presentation / User Manual

Key press inputs:

- W - move camera forward
- S- move camera backward
- A - move camera left
- D - move camera right
- UP - move the duck further on Z-axis
- DOWN - move the duck closer on Z-axis
- LEFT - move the duck left
- RIGHT - move the duck right
- Q - rotate right the duck and well handle
- E - rotate left the duck and well handle
- J - rotate left the directional light

- L - rotate right the directional light
- Y - wireframe mode
- T - solid mode
- O - enable spotlight
- P - enable pointlight
- M - toggle depth pass
- G - generate flower
- Z - start camera animation
- R - enable rain

Figure 1: Initial view of the scene

Figure 2: Spotlight enabled

Figure 3: Point lights enabled

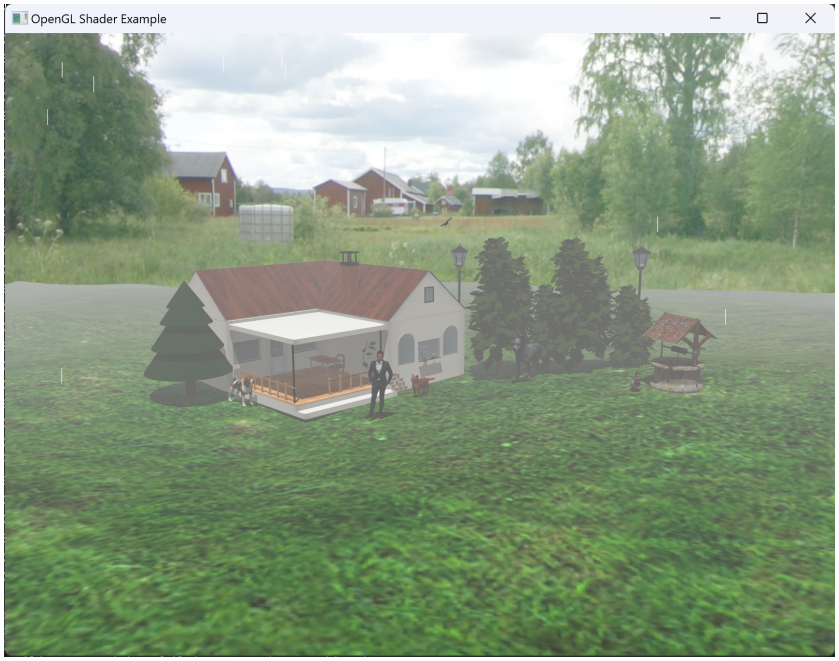

Figure 4: Raining scene

Figure 5: Object generation - flowers

6 Conclusions and Further Developments

In conclusion, this OpenGL project successfully integrates a wide range of techniques to achieve photo-realism, including detailed modeling, advanced algorithms for object generation, collision detection, shadow generation, and environmental effects such as fog, rain, and wind. The scene complexity has been significantly enhanced, showcasing the power of OpenGL in rendering intricate models and realistic animations. Future developments will focus on further refining animation quality, optimizing performance, and enhancing the interaction between dynamic light sources—global, local, and spotlights—within complex environments.

7 References

List all references used in this project, following the appropriate citation style.

- learnopengl.com
- Graphics Processing courses and laboratory works on Moodle