OpenGL Project

Radu Ștefănescu

January 2024

Contents

1 Subject specification

This document presents an overview of a computer graphics project created using OpenGL, depicting a pivotal World War II battle scene. The scene is set in the European theater of war, specifically focusing on a dramatic moment where German forces defend a strategic airfield from an aggressive Soviet tank assault. This project not only aims to recreate a historically significant event with graphical accuracy but also to demonstrate the advanced capabilities of OpenGL in rendering dynamic and complex scenes.

This project not only aims to recreate a historically significant event with graphical accuracy but also to demonstrate the advanced capabilities of OpenGL in rendering dynamic and complex scenes.

Figure 1: Battle Scene

2 Scenario

2.1 Scene and Objects Description

The core of this OpenGL project is an intricately designed scene set in a World War II battlefield, specifically focusing on a German airfield under assault from Soviet forces. The key elements of this scene are meticulously modeled to enhance the realism and historical accuracy of the simulation. The primary objects included in the scene are as follows:

- **Tiger I Tank:** A central piece in the German defense, the Tiger I tank is known for its powerful armor and weaponry. In our scene, it is positioned strategically to defend the airfield. The tank is modeled in detail, capturing its distinctive design features.
- **Jagdtiger Tank Destroyer:** This heavily armored tank destroyer serves as a formidable defensive unit in the scene. The Jagdtiger's imposing presence is recreated with attention to its unique armor and armament specifications.
- **Soviet T-34 Tanks:** Representing the attacking force, two T-34 tanks are included. These are modeled to reflect their historical design, known for mobility and robustness, which made them a staple in the Soviet arsenal.
- **Support Trucks:** Three trucks are positioned in the scene, adding to the authenticity of the wartime environment. These trucks are modeled to resemble the logistical vehicles used during the era.
- **Watchtower with Oil Lamp:** An important aspect of the scene's realism and atmospheric detail is the presence of a watchtower, which overlooks the battlefield.

Figure 2: Tank Shell Hitting T34

2.2 Functionalities

The scene is designed not just for visual representation but also for interactive engagement. The following functionalities have been implemented to enhance the user experience and the dynamism of the scene:

- **Tiger I Turret Control:** Users can control the turret of the Tiger I tank. This includes rotational movement to aim and the ability to fire at targets. This feature is critical for user interaction and engagement with the scene.
- **Projectile Firing Mechanism:** When the Tiger I fires, a projectile is visually rendered, moving towards the target. This feature is crucial for creating a realistic battle scenario.
- **Muzzle Flash Effect (Point Light):** To add to the realism, a muzzle flash effect is implemented each time the Tiger I tank fires. This visual effect enhances the immersion and authenticity of the tank's firing action.
- **Collision Detection and Explosions:** The T-34 tanks are equipped with colliders. When hit by a projectile from the Tiger I, these colliders trigger an explosion effect, signifying the impact and damage. This interaction is vital for creating a dynamic and responsive battle environment.

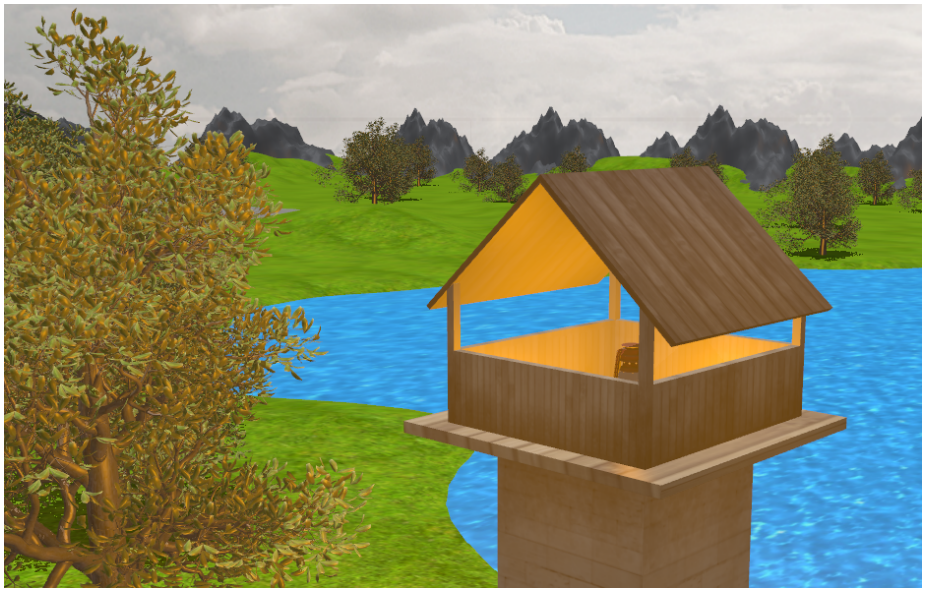

Figure 3: Oil Lamp (Point Light)

3 Implementation details

3.1 Functions and Special Algorithms

One of the key features of this OpenGL project is the camera animation that guides viewers through the scene, showcasing the detailed recreation of the World War II battlefield. This animation is achieved using an interpolation algorithm that smoothly transitions the camera between predefined waypoints. Each waypoint consists of a position and an orientation in 3D space, allowing for a dynamic and cinematic view of the scene. The primary components of this implementation are outlined below:

3.1.1 Waypoint Definition

The camera path is defined using a series of waypoints. Each waypoint consists of a position vector and an orientation vector. The position vector defines the location of the camera in the 3D space, while the orientation vector determines where the camera is looking. Here's the code snippet defining the waypoints:

```
std::vector<Waypoint> cameraPath = {
    // List of waypoints with position and orientation
    // Each waypoint is defined as {{position_x, position_y,
    position_z}, {orientation_x, orientation_y, orientation_z}}
    // Example:
    {{121.037f, 1.93454f, 96.9105f}, {120.352f, 1.91011f, 96.1824f
    }},
    // ... more waypoints ...
};
```

Listing 1: Camera Path Definition

3.1.2 Interpolation Algorithm

The interpolation function is a crucial part of the animation. It calculates the intermediate position and orientation of the camera between two waypoints. The function takes the start and end vectors (position or orientation) and a float t ranging from 0 to 1, representing the interpolation factor. Here's the implementation of the interpolation function:

```
glm::vec3 interpolate(const glm::vec3& start, const glm::vec3& end,
     float t) {
    return start + t * (end - start);
}
```

Listing 2: Interpolation Function

3.1.3 Camera Animation Function

The animateCamera function is responsible for updating the camera's position and orientation over time. It uses the interpolate function to determine the current state of the camera based on the total elapsed time and the defined waypoints. The camera movement speed is controlled by the speed variable, and the function ensures that the animation stops after the last waypoint is reached:

```
void animateCamera(float deltaTime) {
    static float totalTime = 0.0f;
    totalTime += deltaTime;

    float speed = 1.75f; // Speed of the camera movement

    // Calculate the current segment and interpolation factor
    int segment = static_cast<int>(totalTime / speed);
    float t = fmod(totalTime / speed, 1.0f);

    // Stop updating the camera after the last waypoint
    if (segment >= cameraPath.size() - 1) {
        return;
    }

    // Update camera position and orientation
    myCamera.setPosition(interpolate(cameraPath[segment].position,
    cameraPath[segment + 1].position, t));
    myCamera.setCameraTarget(interpolate(cameraPath[segment].
    orientation, cameraPath[segment + 1].orientation, t));
}
```

Listing 3: Animate Camera Function

3.1.4 Rotating the Turret

The turret rotation is implemented by creating a transformation matrix that represents the turret's orientation in the 3D space. This is accomplished by applying a rotation transformation to an identity matrix:

```
glm::mat4 turretRotationMatrix = glm::rotate(glm::mat4(1.0f), glm::
    radians(turretRotationAngle), glm::vec3(0.0f, 1.0f, 0.0f));
```

Listing 4: Turret Rotation Matrix

3.1.5 Firing Mechanism

The firing mechanism simulates the movement of a shell after being fired. The shell's position is updated over time based on its speed and direction:

```
shellTravelDistance += shellSpeed;
shellCurrentPosition = shellStartPosition + shellDirection *
    shellTravelDistance;
```

Listing 5: Shell Movement Update

3.1.6 Collision Detection with Sphere Colliders

Collision detection is performed using sphere colliders. Each tank has a `SphereCollider` struct that defines its collision boundary:

```
struct SphereCollider {
    glm::vec3 center;
    float radius;
};
```

Listing 6: Sphere Collider Structure

The collision check is conducted in the `checkCollision` function. It calculates the distance between the shell's position and the collider's center, detecting a collision if the distance is less than the sum of the shell's and collider's radii:

```
float distance = glm::distance(collider.center, glm::vec3(
    rotatedShellPosition));
return distance < collider.radius + 1.25f;
```

Listing 7: Collision Detection Function

3.2 Graphics Model

3.2.1 Fog Implementation

Fog is a graphical effect in 3D rendering that enhances the realism of a scene by simulating the attenuation of light over distance. It is often used to create a sense of depth and atmosphere. In this OpenGL project, the implementation of fog plays a crucial role in conveying the environmental conditions typical of a World War II battlefield. The fog effect is achieved by gradually blending the colors of objects with the fog color based on their distance from the camera. This not only adds a layer of depth to the scene but also helps in setting a tone that aligns with the historical and environmental context of the battle. The adjustable parameters of fog density and color allow for simulating different weather conditions, further enhancing the immersive experience.

Figure 4: Fog

3.2.2 Shadow Mapping

Shadow mapping is an essential technique in 3D graphics for creating realistic shadows, crucial for adding depth and a sense of realism to the scene. In this project, shadow mapping is implemented to enhance the visual fidelity of the battlefield. This technique involves rendering the scene from the light's perspective to create a depth map, which is then used to determine which areas are shadowed from the main camera's viewpoint. Accurate shadow mapping contributes significantly to the realism of the scene, especially in emphasizing the dimensions and physical presence of objects like tanks, buildings, and terrain features. The interplay of light and shadow is meticulously calibrated to reflect the time of day and weather conditions depicted in the scene, adding a dynamic and engaging element to the visualization.

Figure 5: Enter Caption

3.3 Data Structures

This OpenGL project utilizes several key data structures, each playing a crucial role in the rendering and interaction within the 3D environment.

3.3.1 SphereCollider and Waypoint Structures

The `SphereCollider` structure is integral for collision detection. It comprises a `glm::vec3` for the sphere's center and a float for its radius, facilitating efficient collision checks. This structure is particularly useful in scenarios like shell collision with tanks.

The `Waypoint` structure, essential for camera animation, consists of two `glm::vec3` objects representing a camera's position and orientation at specific scene points. These waypoints create a path, enabling smooth camera movement and dynamic viewing angles through interpolation.

3.3.2 GLM Matrices and Vectors

The project extensively employs GLM (OpenGL Mathematics) library's matrices and vectors for 3D transformations and calculations. `glm::mat4` is used for model, view, and projection matrices, fundamental in 3D rendering. These matrices manage object transformations, camera setup, and 3D-to-2D scene projection. Vectors like `glm::vec3` and `glm::vec4` represent 3D points and directions, crucial for defining object positions, light directions, and camera movements.

3.4 Class Hierarchy

The class hierarchy of this OpenGL project is designed to foster a modular and scalable architecture, essential for maintaining the complexity inherent in 3D rendering and interaction.

At the core, abstract base classes define general behaviors and properties, which are then extended by more specific subclasses. This approach encapsulates shared functionalities (like rendering, transformations, and basic geometrical calculations) within parent classes, while allowing subclasses to introduce more detailed, context-specific features (such as unique behaviors of different types of game objects, specialized shaders, or particular physics calculations).

In essence, the hierarchy ranges from general to specific, ensuring both code reusability and adaptability. Classes like `Model3D`, `Camera`, and `Shader` serve as foundational building blocks, each encapsulating key aspects of the graphics rendering pipeline. This structured approach not only streamlines the development process but also enhances the maintainability and extendibility of the codebase, accommodating future enhancements or modifications with ease.

4 Graphical User Interface Presentation / User Manual

This section provides an overview of the user interface and control scheme for the OpenGL project, designed to offer an intuitive and interactive experience.

4.1 Camera Control

- **Mouse Movement:** The camera's view direction is controlled by moving the mouse. This allows for fluid and precise adjustments to the viewing angle.
- **WASD Keys:** These keys are used to move the camera through the scene:
  - **W:** Move forward
  - **A:** Move left
  - **S:** Move backward
  - **D:** Move right

4.2 Rendering Modes

- **T Key:** Switch to wireframe mode.
- **Y Key:** Switch to solid mode.
- **U Key:** Switch to points mode.

4.3 Interactive Features

- **I Key:** Turn on/off the oil lamp in the watchtower.
- **O Key:** Toggle fog effect in the scene.
- **Z and X Keys:** Rotate the turret left or right.
- **K Key:** Fire the tank's shell.
- **M Key:** Toggle the depth map view.

This control scheme is designed to be user-friendly, providing an immersive and interactive experience in exploring and interacting with the 3D environment.

5 Conclusions and Further Developments

In conclusion, this OpenGL project successfully demonstrates the power and versatility of modern 3D graphics programming. By recreating a World War II scene with interactive elements like tank turret control, projectile firing, and environmental effects such as fog and dynamic lighting, the project not only showcases technical proficiency but also offers a platform for immersive historical exploration. The use of advanced rendering techniques like shadow mapping and the implementation of intuitive controls further enhances the user experience.

5.1 Further Developments

For future developments, potential enhancements could include the integration of more complex physics simulations for more realistic interactions, the implementation of AI for dynamic enemy movement, and the expansion of the environment to include more varied terrain and additional historical scenarios. Additionally, the introduction of networked multiplayer capabilities could transform the experience into an interactive educational tool or a collaborative exploration platform.

6 References

1. YouTube
2. Wikipedia