OpenGL Project Documentation

Student: Butas Rafael-Dorian
Coordinator: Constantin Ioan Nandra

13 January 2025

Contents

1 Subject Specification

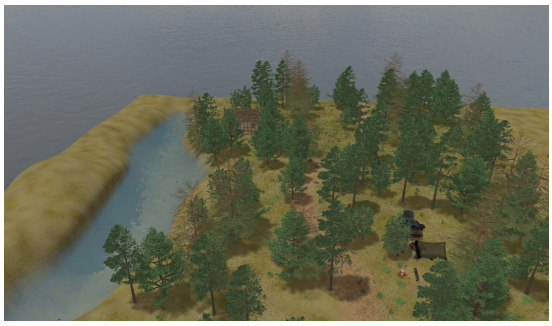
The objective of this project is to develop a highly realistic and interactive 3D forest environment using modern OpenGL. It aims to demonstrate advanced real-time rendering capabilities through sophisticated graphics techniques (e.g., HDR, Bloom, Shadow Mapping) and dynamic environmental effects (e.g., Rain, Fog, Wind). The project integrates asset creation, scene composition, and performance optimizations to deliver an immersive virtual experience with detailed visuals, interactive controls, and synchronized audio.

2 Scenario

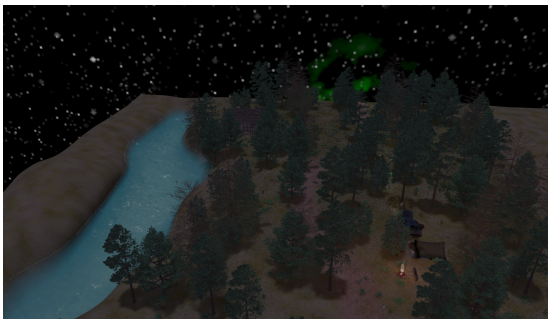
2.1 Scene and Objects Description

The 3D forest scene blends natural and man-made elements to create an engaging outdoor environment. Key elements include:

- **Forest Environment:** A dense collection of trees, grass, plants, ferns, and rocks spread over varied terrain, intersected by a main road.



(a) Daytime View



(b) Nighttime View

Figure 1: The 3D forest environment showcasing all key elements including the campsite, barn, and lake, during different times of day.

- **Camping Site:** This area includes a campfire, wooden bench, fishing rod, food items, and a tent with various camping accessories:

Figure 2: Campsite with campfire, seating, and other camping equipment.

- **Barn Area:** A rustic barn near the lake contains hay, a ladder, crates, barrels, and other stored items:

Figure 3: Barn area featuring hay, a ladder, crates, and barrels next to the lake.

- **Lake:** A serene lake adjacent to the barn, featuring dynamic water rendering:

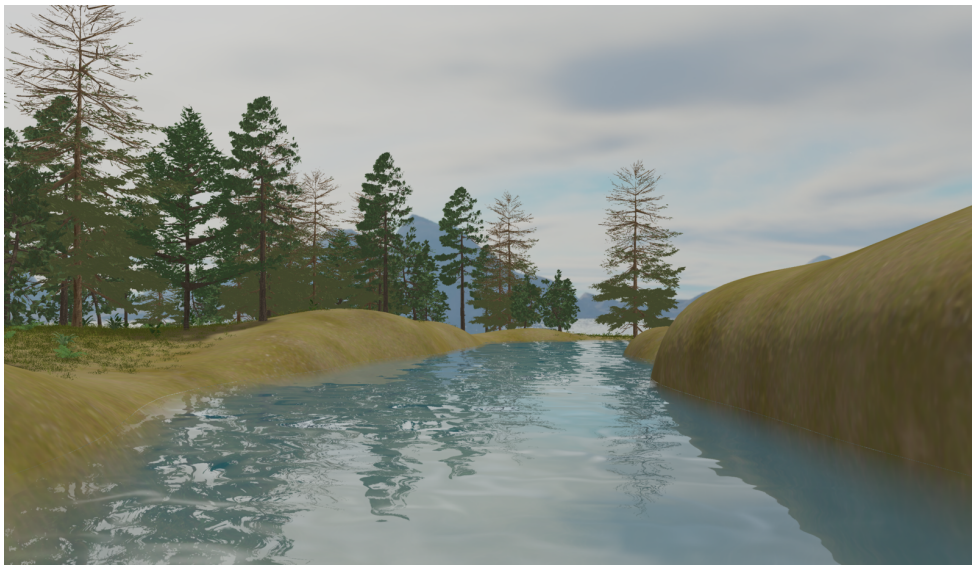


Figure 4: Lake reflecting surrounding environment.

Overall, the combination of natural foliage, campsite furnishings, and a rustic barn creates an immersive setting, enriched by day/night transitions and various interactive elements.

2.2 Functionalities

This project implements a comprehensive set of features for realistic and interactive 3D rendering, broadly categorized into the following:

2.2.1 Rendering Techniques

- **Dynamic Lighting:**
  - *Directional Light:* Global illumination simulating sun/moon, complete with shadow maps.
  - *Point Light:* Localized sources like a campfire, casting omnidirectional shadows.
  - *Spotlight:* Focused beams (e.g., flashlight/headlights) with cone-shaped shadows.

  - *Blinn-Phong Model:* Combines ambient, diffuse, and specular reflections for realistic shading.

- **Shadow Mapping:**
  - Depth-map–based shadows for directional, point, and spotlights, using PCF for soft edges.
- **Normal Mapping:**
  - Per-pixel surface detail using tangent-bitangent-normal (TBN) transformations.
- **HDR and Bloom Effects:**
  - HDR extends the color range; Bloom adds a glow around bright regions.
- **Fragment Discarding Effect:**
  - Uses dissolve (alpha) textures for selective pixel discarding, enabling partial transparency.
- **Water Effect (Lake Rendering):**
  - Reflective/refractive water surface with wave animations, fresnel-based blending, and normal maps.
- **Rain-Integrated Shading:**
  - Blinn-Phong shading on wet surfaces during rainfall, emphasizing highlights and water accumulation.

2.2.2 Environmental Effects

- **Rain Simulation:**
  - GPU-based raindrop particles with motion blur and dynamic lighting interactions.
- **Fog Effect:**
  - Layered fog and animated swirling fog, adjusting density and color based on time or weather.
- **Wind Effect:**
  - Perlin-noise–driven wind motion affecting foliage, with adjustable strengths and scales.
- **Fire Effect:**
  - Flame, smoke, and embers rendered as GPU-instanced particles near the campfire.
- **Lightning Effect:**
  - Flashlight-based lightning flashes with randomized intervals, plus thunder audio delay.

2.2.3 Audio Integration

- **Environmental Sounds:**
  - Rain loops, thunder claps (with random delay), and fire crackling sounds, each spatially or ambiently configured.

2.2.4 User Interaction

- **Camera Navigation:**
  - Free-fly camera with `WASD` movement, vertical controls, and mouse-based yaw/pitch.
  - Waypoint-based tour system (record, save, load) for guided exploration.
- **Graphical User Interface Controls:**
  - Keyboard inputs to toggle rendering modes, lighting, and effects in real-time.
  - Console feedback logs confirm toggles (e.g., `Bloom Toggled: ON`).

2.2.5 Optimization Strategies

- **Frustum Culling + BVH:**
  - Bounding volume hierarchy for rapid visibility checks, skipping off-screen geometry.
  - Frustum culling at the node/object level to avoid unnecessary draw calls.
- **Mesh Batching:**
  - Consolidates geometry with similar materials, reducing state changes.
  - Splits large batches to maintain effective culling.
- **Texture and Geometry Optimization:**
  - Downscaled textures for repeated assets, simplified meshes via `meshoptimizer` to maintain performance.
- **Shader Optimizations:**
  - Uniform caching and minimal redundant lookups for better GPU efficiency.

2.2.6 Advanced Rendering Features

- **Skybox:**
  - Surrounds the scene with a dynamic sky, blending day/night transitions smoothly.
- **Procedural and Dynamic Elements:**
  - Procedural terrain or vegetation instancing (potential future improvement).

2.2.7 Real-Time Post-Processing Effects

- **Tone Mapping:**
  - Converts HDR color ranges to a display-friendly format.
- **Gaussian Blur for Bloom:**
  - Multi-pass blur to add a subtle glow around bright areas.

These integrated functionalities combine to form a visually striking and highly interactive 3D forest environment. By leveraging advanced rendering techniques, dynamic effects, synchronized audio, and robust optimization, the system achieves both high fidelity and real-time performance.

3 Implementation Details

3.1 Lighting Model in OpenGL

The lighting implementation in this project is designed to create realistic and dynamic lighting effects in the 3D forest scene. It utilizes multiple types of lights, integrated with the Blinn-Phong reflection model, to enhance visual realism and interactivity.

3.1.1 Directional Light

- **Purpose:** Simulates sunlight or moonlight, providing global illumination for the scene.
- **Implementation:** The directional light (`DirLight`) casts uniform parallel rays across the scene. Its ambient, diffuse, and specular components determine the overall brightness, color, and highlights. Shadows are calculated using shadow mapping techniques to simulate realistic occlusion caused by objects.

3.1.2 Point Light

- **Purpose:** Represents localized light sources, such as the campfire in the scene, which emit flickering light in all directions.
- **Implementation:** The point light (`PointLight`) uses attenuation factors (`constant`, `linear`, and `quadratic`) to simulate realistic light decay over distance. Its ambient and diffuse components interact with nearby objects, enhancing the natural glow effect of the campfire.

3.1.3 Spotlight

- **Purpose:** Models dynamic and focused light sources, such as a flashlight held by the player, illuminating a cone-shaped area.
- **Implementation:** The spotlight (`SpotLight`) is controlled dynamically, allowing its position and direction to change in real-time. The light's cutoff angles define the core and falloff regions, creating a realistic fade effect at the cone's edge. This enables targeted illumination for exploration and interaction within the forest.

3.1.4 Headlights

- **Purpose:** Simulates car headlights (static spotlights) to illuminate the road ahead and nearby areas dynamically.
- **Implementation:** Two spotlights (`leftHeadlight` and `rightHeadlight`) are positioned relative to the car model. These headlights feature soft edges and shadows to enhance the nighttime ambiance and visibility in the scene, especially along the forest road.

3.1.5 Blinn-Phong Reflection Model and Texture Integration

- **Overview:** The Blinn-Phong model is employed for shading calculations. It combines three components:
  - **Ambient:** Simulates indirect light scattered uniformly across surfaces.
  - **Diffuse:** Calculates light intensity based on the angle between the light direction and the surface normal.
  - **Specular:** Simulates highlights caused by reflected light, using a half-vector to improve performance and accuracy over the Phong model.
- **Textures in Lighting:**
  - **Diffuse Texture:** The diffuse texture defines the base color of the surface and interacts with the ambient and diffuse lighting components to produce realistic surface shading.
  - **Specular Texture:** The specular texture determines the intensity and pattern of the reflective highlights on a surface, enhancing material realism.
- **Rain Simulation:** During rainy conditions, the Blinn-Phong model is combined with specular textures to simulate the appearance of wet vegetation and other surfaces. The specular highlights are intensified to reflect the glistening effect of water on surfaces, adding to the scene's immersive realism.
- **Integration:** Each light source contributes its ambient, diffuse, and specular components to the final illumination. These are combined with the textures to ensure that the surfaces respond realistically under varying lighting conditions.

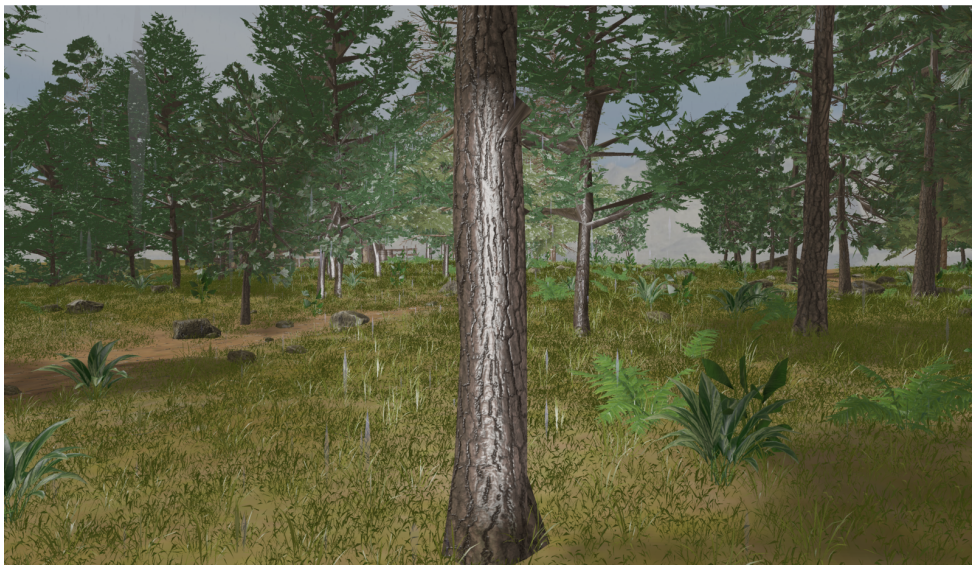


Figure 5: Blinn-Phong Reflection Model While Raining: Enhanced specular highlights on wet surfaces.

3.2 Camera Navigation

Camera navigation is a core feature of the project, allowing users to explore the 3D forest scene interactively and intuitively. The camera system includes multiple functionalities such as free movement, rotation, zoom, and a pre-recorded tour system for an immersive experience.

3.2.1 Camera Movement and Orientation

The camera can be moved freely within the scene, providing the ability to navigate the forest environment from various perspectives.

- **Movement Directions:**
  - The camera supports movement in six directions: forward, backward, left, right, up, and down.
  - Movement is relative to the camera's current orientation, ensuring natural navigation.
- **Rotation:**
  - Users can rotate the camera around its pitch (vertical) and yaw (horizontal) axes to look in different directions.
  - Mouse input is used for smooth and precise camera rotation.
- **Zoom:**
  - The camera's field of view (FOV) can be adjusted using the mouse scroll wheel to zoom in and out.
  - This enhances the ability to inspect objects closely or view the scene from a wider perspective.

3.2.2 Pre-Recorded Tour System

The pre-recorded tour system is a pivotal feature of the project, enabling users to experience a guided exploration of the 3D forest environment. This system meticulously records specific camera positions and orientations, interpolates smooth transitions between these points, and autonomously navigates the camera along the predefined path. The implementation leverages Catmull-Rom spline interpolation and easing functions to ensure fluid and cinematic movements.

Waypoint Recording Waypoint recording allows users to mark specific locations within the scene that the camera should visit during the tour. Each waypoint captures the camera's **position**, **target**, and **up direction**, encapsulating the complete state required for rendering the scene from that viewpoint.

- **User Interaction:**
  - **Recording Waypoints:** Users can press the `P` key to record the current camera state. This action invokes the `recordWaypoint()` function, which retrieves the camera's current position, target, and up direction using `myCamera.getPosition()`, `myCamera.getTarget()`, and `myCamera.getUpDirection()`, respectively.
  - **Saving Waypoints:** Pressing the `O` key triggers the `saveWaypoints("waypoints.txt")` function, which writes all recorded waypoints to a file named `waypoints.txt`. This ensures that users can persist their tours and reuse them in future sessions.
  - **Loading Waypoints:** By pressing the `L` key, users can load previously saved waypoints from `waypoints.txt` using the `loadWaypoints("waypoints.txt")` function. This function reads each line from the file, parses the camera states, and populates the `keyLocations` vector with `Location` structures containing position, target, and up direction data.
  - **Removing Waypoints:** The `Minus (-)` key allows users to remove the last recorded waypoint, providing flexibility to adjust the tour path as needed.
- **Data Structure:**
  - **Location Structure:** Each waypoint is represented by a `Location` structure comprising three `glm::vec3` vectors: `position`, `target`, and `up`. These vectors precisely define the camera's state at each waypoint.
  - **Storage:** All waypoints are stored in a `std::vector<Location>` named `keyLocations`, facilitating dynamic management and easy access during the tour.

Path Interpolation To ensure smooth and natural transitions between waypoints, the system employs **Catmull-Rom spline interpolation** combined with an **easing function**. This combination allows the camera to move fluidly along the path, avoiding abrupt changes in direction or speed.

- **Catmull-Rom Spline Interpolation:**
  - **Purpose:** Catmull-Rom splines provide a method for generating smooth curves that pass through a series of control points (waypoints). This ensures that the camera follows a natural path through the environment.
  - **Implementation:** The `catmullRomInterpolate()` function computes interpolated positions between four consecutive waypoints (`P0`, `P1`, `P2`, `P3`) based on a parameter `t` ranging from 0 to 1. This parameter represents the progression along the spline between `P1` and `P2`.

Listing 1: Catmull-Rom Spline Interpolation Function

```
glm::vec3 catmullRomInterpolate(const glm::vec3& P0, const glm::vec3& P1,
    const glm::vec3& P2, const glm::vec3& P3, float t) {
    return 0.5f * (
        (2.0f * P1) +
        (-P0 + P2) * t +
        (2.0f * P0 - 5.0f * P1 + 4.0f * P2 - P3) * t * t +
        (-P0 + 3.0f * P1 - 3.0f * P2 + P3) * t * t * t
        );
}
```

- **Easing Function:**
  - **Purpose:** The `easeInOut()` function modifies the interpolation parameter `t` to control the pacing of the camera movement. This function ensures that the camera accelerates smoothly at the beginning of the transition and decelerates as it approaches the next waypoint, mimicking natural motion.

Listing 2: Ease-In-Out Function

```
float easeInOut(float t) {
    return t < 0.5f ? 2.0f * t * t : -1.0f + (4.0f - 2.0f * t) * t;
}
```

- **Transition Management:**
  - **Blend Factors:** Variables such as `blend`, `transitionDuration`, `transitionStartTime`, `startBlend`, and `targetBlend` manage the blending between different states or effects during transitions, ensuring visual consistency and smoothness.

Listing 3: Transition Variables

```
    // Transition variables
    float blend = 1.0f;
    float transitionDuration = 3.0f;
    float transitionStartTime = 0.0f;
    float startBlend = 1.0f;
    float targetBlend = 1.0f;
```

Automatic Navigation Automatic navigation orchestrates the camera's movement along the interpolated path, providing a seamless and cinematic tour experience. This process involves iterating through waypoints, applying interpolation, and updating the camera's state accordingly.

- **Tour Activation:**
  - **Starting the Tour:** When the user presses the `T` key, the tour is activated by setting `isTourActive` to `true`, resetting `currentWaypoint` to `0`, and initializing the interpolation parameter `t` to `0.0f`.
  - **Stopping the Tour:** Pressing the `Y` key deactivates the tour by setting `isTourActive` to `false`, allowing users to regain manual control over the camera.

Listing 4: Key Press Handling for Tour Control

```
    if (key == GLFW_KEY_T) {
        isTourActive = true;
        currentWaypoint = 0;
        t = 0.0f;
        std::cout << "Tour started" << std::endl;
    }
    if (key == GLFW_KEY_Y) {
        isTourActive = false;
        std::cout << "Tour stopped manually" << std::endl;
    }
```

- **Update Mechanism:** The `updateTour()` function is called every frame to advance the camera along the tour path. It checks if the tour is active and ensures that there are enough waypoints to perform interpolation.

Listing 5: Update Tour Function

```
void updateTour() {
    if (!isTourActive || currentWaypoint >= keyLocations.size() - 3) return
        ;
    size_t idx = currentWaypoint;
    if (idx + 3 >= keyLocations.size()) {
        std::cerr << "Error: Attempting to access keyLocations out of
            bounds." << std::endl;
        isTourActive = false;
        return;
    }
    const Location& P0 = keyLocations[idx];
```

```
10      const Location& P1 = keyLocations[idx + 1];
11      const Location& P2 = keyLocations[idx + 2];
12      const Location& P3 = keyLocations[idx + 3];
13
14      float easedT = easeInOut(t);
15
16      glm::vec3 interpolatedPos = catmullRomInterpolate(P0.position, P1.
            position, P2.position, P3.position, easedT);
17      glm::vec3 interpolatedTarget = catmullRomInterpolate(P0.target, P1.
            target, P2.target, P3.target, easedT);
18      glm::vec3 interpolatedUp = catmullRomInterpolate(P0.up, P1.up, P2.up,
            P3.up, easedT);
19      glCheckError();
20
21      myCamera.setPosition(interpolatedPos);
22      myCamera.setTarget(interpolatedTarget);
23      myCamera.setUpDirection(interpolatedUp);
24
25      glm::mat4 view = myCamera.getViewMatrix();
26      glCheckError();
27
28      myBasicShader.useShaderProgram();
29      glCheckError();
30      glUniformMatrix4fv(basicUniforms.view, 1, GL_FALSE, glm::value_ptr(view
            ));
31      glCheckError();
32
33      t += tIncrement;
34      if (t > 1.0f) {
35          t = 0.0f;
36          currentWaypoint++;
37          if (currentWaypoint >= keyLocations.size() - 3) {
38              isTourActive = false;
39              std::cout << "Tour completed" << std::endl;
40          }
41      }
42  }
```

- **Interpolation Steps:**
  1. **Current Segment Identification:** The system identifies the current segment of the path by selecting four consecutive waypoints (`P0`, `P1`, `P2`, `P3`). This quartet is essential for Catmull-Rom spline interpolation.
  2. **Easing Application:** The interpolation parameter `t` is modified using the `easeInOut()` function to control the pacing of movement.
  3. **Position and Orientation Interpolation:** The camera's position, target, and up direction are interpolated separately using the Catmull-Rom function, ensuring that all aspects of the camera's state transition smoothly.
  4. **Camera State Update:** The interpolated values are applied to the camera using `myCamera.setPosition()`, `myCamera.setTarget()`, and `myCamera.setUpDirection()`.
  5. **View Matrix Update:** The view matrix is recalculated based on the new camera state and passed to the shader program to reflect the updated perspective.
  6. **Progression Control:** The interpolation parameter `t` is incremented by `tIncrement` each frame. Once `t` exceeds `1.0f`, it resets to `0.0f`, and `currentWaypoint` advances to the next segment. If the tour reaches the end of the waypoints, it automatically deactivates.
- **Error Handling:** The implementation includes safeguards to prevent out-of-bounds access to the `keyLocations` vector. If an invalid waypoint index is detected, the tour is halted, and an error message is logged.

Listing 6: Error Handling in Tour Update

```
if (idx + 3 >= keyLocations.size()) {
    std::cerr << "Error: Attempting to access keyLocations out of bounds."
        << std::endl;
    isTourActive = false;
    return;
}
```

- **Initialization:** Upon starting the application, the system attempts to load existing waypoints from `waypoints.txt`. This ensures that users can immediately begin tours based on previously saved paths without manual setup.

Listing 7: Waypoint Initialization in Main Function

```
int main() {
    // Other initialization code...

    // Load waypoints from file at startup
    loadWaypoints("waypoints.txt");

    // Rest of the main loop...
}
```

Integration with Scene Rendering The pre-recorded tour system seamlessly integrates with the scene's rendering pipeline to ensure that camera movements are accurately reflected in the rendered visuals.

- **View and Projection Matrices:**
  - **View Matrix:** Updated dynamically based on the interpolated camera position, target, and up direction. This matrix determines the camera's perspective within the scene.
  - **Projection Matrix:** Adjusted to accommodate changes in the camera's field of view (FOV) during zoom operations, maintaining consistency in the rendered output.

Listing 8: View Matrix Update in Update Tour

```
glm::mat4 view = myCamera.getViewMatrix();
glUniformMatrix4fv(basicUniforms.view, 1, GL_FALSE, glm::value_ptr(view));
```

- **Shader Synchronization:**
  - The updated view matrix is passed to the shader program (`myBasicShader`) each frame, ensuring that all objects and effects are rendered from the correct camera perspective.

Listing 9: Shader Program Usage and Uniform Update

```
myBasicShader.useShaderProgram();
glUniformMatrix4fv(basicUniforms.view, 1, GL_FALSE, glm::value_ptr(view));
```

- **Cinematic Experience:**
  - By interpolating both the camera's position and orientation, the system creates a cohesive and engaging narrative flow. Transitions between waypoints are smooth, preventing jarring jumps and maintaining the immersion of the user.

Implementation Summary The pre-recorded tour system is meticulously crafted to offer a smooth and immersive guided exploration of the 3D forest environment. By leveraging Catmull-Rom spline interpolation and easing functions, the system ensures that camera movements are both natural and cinematic. User-friendly controls facilitate easy recording, saving, loading, and management of waypoints, while seamless integration with the rendering pipeline guarantees that the visual output accurately reflects the camera's journey. Robust error handling and initialization processes further enhance the system's reliability and user experience.

3.2.3 Integration with Scene Rendering

The camera's position and orientation are seamlessly integrated with the scene rendering pipeline:

- **View Matrix:**
  - The camera's view matrix is updated dynamically based on its position and orientation.
  - This ensures that the scene is rendered correctly from the camera's perspective.
- **Projection Matrix:**
  - The projection matrix adjusts dynamically to reflect changes in the camera's FOV during zoom operations.
  - All shaders, including those for objects, rain, and the skybox, are updated with the latest view and projection matrices.

3.2.4 User Interaction and Control

The camera system provides an intuitive interface for user control:

- **Keyboard Input:**
  - Keys `W`, `A`, `S`, `D`, `Q`, and `E` allow movement in respective directions.
- **Mouse Input:**
  - Mouse movement controls pitch and yaw for free rotation.
  - The scroll wheel adjusts zoom by changing the FOV.
- **Tour Activation:**
  - Users can start the pre-recorded tour, pausing manual controls while the camera follows the tour path.

The camera navigation system is a fundamental component of the project, enabling users to explore the 3D forest interactively or enjoy a guided tour that highlights its immersive environment.

3.3 Shadow Mapping

Shadow mapping adds depth and realism by determining which fragments in the scene are in shadow relative to a particular light source. The core idea is to render the scene from the perspective of the light and store depth information in a texture (the *shadow map*). During the final rendering pass, each fragment's depth is compared to the stored depth to decide whether that fragment is occluded from the light or not.

In this project, we create separate shadow maps for:

- **Directional Light** (e.g., the sun or moon)
- **Point Light** (e.g., a campfire)
- **Spotlights** (e.g., headlights or flashlights)

The following subsections detail how each shadow map is set up, the shaders involved, and how the shadows are integrated back into the final scene render.

3.3.1 Directional Light Shadows

Purpose Simulates shadows cast by large, distant sources (e.g., the sun or moon). Because these lights can be approximated as infinitely far away, we use an *orthographic projection* to capture the entire scene in the light's frustum.

Implementation

1. **Depth Map Generation:**
   - A dedicated framebuffer (`shadowMapFBO`) is created for directional light.
   - A single 2D depth texture (`depthMap`) of resolution 4096×4096 is attached as a depth attachment.
   - No color attachment is used (color buffers are not needed for shadows).
   - This pass is carried out in the function `renderDepthMap()`, which:
     - Computes an orthographic projection matrix (stored in `lightProjection`), based on the size of the scene.
     - Computes a `lightView` matrix using `glm::lookAt()`, placing the camera at the light's position, looking toward the origin.
     - Multiplies them to get `lightSpaceMatrix = lightProjection * lightView`.
     - Binds `shadowMapFBO` and clears its depth buffer.
     - Renders all objects using the `shadowShader`.
2. **Vertex Shader (Directional):**
   - Key lines:

     ```
     gl_Position = lightSpaceMatrix * model * vec4(pos, 1.0);
     ```

   - Optional wind perturbations are applied if `windEnabled == 1` and the object is marked as `isWindMovable`.
   - The wind function uses Perlin Noise to create realistic motion in vegetation.
3. **Fragment Shader (Directional):**
   - In this project, the directional shadow fragment shader is mostly empty (or simply writes depth via `gl_FragDepth` by default).
   - By default, OpenGL uses the depth value of `gl_Position` to populate the depth texture.

Sampling the Shadow Map During the final rendering pass (using `main fragment shader`), we compare each fragment's depth (in light space) to the stored `depthMap`. This occurs in:

```
float ShadowCalculation(vec3 normal, vec4 fragPosLightSpace,
                        vec3 fragPos, vec3 lightDir) { ... }
```

If `fragPosLightSpace.z` is greater than the depth sampled from the shadow map, the fragment is considered in shadow. *Percentage Closer Filtering (PCF)* is performed by sampling multiple nearby texels to soften shadow edges.

3.3.2 Point Light Shadows

Purpose Point lights emit light in all directions, requiring an *omnidirectional* shadow solution. We render depth from six directions (forming a cube map) around the light source (e.g., a campfire).

Implementation

1. **Depth Cube Map Generation:**
   - A dedicated framebuffer (`pointLightFBO`) is created for point lights.
   - A cube map texture (`depthCubemap`) with resolution 1024×1024 is attached as a depth attachment (`GL_TEXTURE_CUBE_MAP`).
   - In the function `renderDepthCubemap()`, we:
     - Use a perspective projection of 90° (`shadowProj`) for each cube face.
     - For each face of the cube map, create a `lookAt` matrix pointing along $(\pm x, \pm y, \pm z)$.

- Store all these matrices in `shadowMatrices[i]`.
- Render the scene from each of the six directions using `pointShadowShader`.

2. **Vertex + Geometry + Fragment Shaders (Point):**

   - **Vertex Shader (Point)** is similar to the directional vertex shader but outputs positions to be used later in the geometry shader.
   - **Geometry Shader:**

Listing 10: Point Light Geometry Shader

```
layout (triangles) in;
layout (triangle_strip, max_vertices=18) out;

uniform mat4 shadowMatrices[6];
out vec4 FragPos;
...
for(int face = 0; face < 6; ++face)
{
    gl_Layer = face;
    for(int i = 0; i < 3; ++i)
    {
        FragPos = gl_in[i].gl_Position;
        gl_Position = shadowMatrices[face] * FragPos;
        EmitVertex();
    }
    EndPrimitive();
}
```

   - **Fragment Shader (Point)**:

Listing 11: Point Light Fragment Shader

```
uniform vec3 lightPos;
uniform float far_plane;

void main()
{
    float lightDistance = length(FragPos.xyz - lightPos);
    // normalize distance by far_plane
    lightDistance = lightDistance / far_plane;
    gl_FragDepth = lightDistance;
}
```

This stores a normalized distance from the light in the cube map's depth buffer.

Sampling the Point Light Shadow Map In the final pass, we retrieve the distance from the fragment to the point light and compare it to the value stored in the cube map. This is handled by:

```
float PointShadowCalculation(vec3 normal, vec3 fragPos, vec3 lightDir) { ... }
```

A set of 20 offset directions (`sampleOffsetDirections`) is used to do multiple texture lookups around the fragment's position in the cube map. This filters the shadow and reduces artifacts:

```
for (int i = 0; i < samples; ++i)
{
    float closestDepth = texture(pointLightShadowMap,
                 fragToLight + sampleOffsetDirections[i] * diskRadius).r;
    // ...
}
```

The result is averaged to soften shadow edges.

3.3.3 Spotlight Shadows

Purpose Spotlights (e.g., car headlights or a player's flashlight) cast focused cone-shaped shadows. A *perspective projection* is used to capture only the cone region in front of the spotlight.

Implementation

1. **Framebuffer + Texture Setup:**
   - Each spotlight has its own FBO and 2D depth texture (e.g., `leftHeadlightFBO, leftHeadlightDepthMap`).
   - Resolution typically 1024×1024 for each spotlight.
2. **Depth Map Render Pass:**
   - For a spotlight (e.g., `leftHeadlight`), we form `lightProjectionHead` using a perspective projection (FOV about 60°), combined with a `lightViewHead` from the spotlight's position along its direction.
   - The final matrix:

     ```
     glm::mat4 lightSpaceMatrixHead =
                     lightProjectionHead * lightViewHead;
     ```

   - We use `headShadowShader` for rendering the scene into the depth texture. This is nearly identical to the directional approach but uses a perspective projection.
   - Wind calculations are also present in the vertex shader (same Perlin Noise approach).

Sampling the Spotlight Shadow Map During the final pass, we compare the fragment's position in the spotlight's light space against the stored depth using:

```
float HeadlightShadowCalculation(
    vec3 normal, vec4 fragPosLightSpace,
    vec3 fragPos, vec3 lightDir,
    sampler2D headlightShadowMap) { ... }
```

We again apply a small PCF search to soften edges.

3.3.4 Framebuffer and Texture Setup (Code Details)

Below is a concise mapping of how the framebuffers and textures are set up in C++:

- **Directional Light FBO:**

Listing 12: Directional Light FBO Setup

```
glGenFramebuffers(1, &shadowMapFBO);
glBindFramebuffer(GL_FRAMEBUFFER, shadowMapFBO);

glGenTextures(1, &depthMap);
glBindTexture(GL_TEXTURE_2D, depthMap);
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT,
    SHADOW_WIDTH, SHADOW_HEIGHT, 0,
    GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
// Set texture parameters: filtering, wrap mode, border color, etc.

glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
                       GL_TEXTURE_2D, depthMap, 0);
glDrawBuffer(GL_NONE);
glReadBuffer(GL_NONE);

glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

- **Point Light FBO (Cube Map):**

Listing 13: Point Light FBO (Cube Map) Setup

```
glGenFramebuffers(1, &pointLightFBO);
glBindFramebuffer(GL_FRAMEBUFFER, pointLightFBO);

glGenTextures(1, &depthCubemap);
glBindTexture(GL_TEXTURE_CUBE_MAP, depthCubemap);
for (unsigned int i = 0; i < 6; ++i) {
    glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i,
        0, GL_DEPTH_COMPONENT,
        POINT_SHADOW_WIDTH, POINT_SHADOW_HEIGHT, 0,
        GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
}
// Set texture parameters (MIN_FILTER, MAG_FILTER, CLAMP_TO_EDGE, etc.)

glFramebufferTexture(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
                     depthCubemap, 0);
glDrawBuffer(GL_NONE);
glReadBuffer(GL_NONE);

glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

- **Spotlight FBO (Headlights):**

Listing 14: Spotlight FBO (Headlights) Setup

```
glGenFramebuffers(1, &FBO);
glBindFramebuffer(GL_FRAMEBUFFER, FBO);

glGenTextures(1, &depthMap);
glBindTexture(GL_TEXTURE_2D, depthMap);
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT,
    SPOT_LIGHT_SHADOW_WIDTH, SPOT_LIGHT_SHADOW_HEIGHT,
    0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
// Set texture parameters (filtering, wrap mode, border color, etc.)

glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
                       GL_TEXTURE_2D, depthMap, 0);
glDrawBuffer(GL_NONE);
glReadBuffer(GL_NONE);

glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

3.3.5 Integration with Scene Rendering

Binding Shadow Maps Once the shadow maps are generated for each light (directional, point, and each spotlight), they are bound and passed to the main shader program:

Listing 15: Binding Shadow Maps to the Main Shader

```
glActiveTexture(GL_TEXTURE0 + 4);
glBindTexture(GL_TEXTURE_2D, depthMap);
glUniform1i(basicUniforms.shadowMap, 4);

// Point light shadow map
glActiveTexture(GL_TEXTURE0 + 5);
glBindTexture(GL_TEXTURE_CUBE_MAP, depthCubemap);
glUniform1i(basicUniforms.pointLightShadowMap, 5);

// Left headlight shadow map
glActiveTexture(GL_TEXTURE0 + 6);
glBindTexture(GL_TEXTURE_2D, leftHeadlightDepthMap);
glUniform1i(basicUniforms.leftHeadlightShadowMap, 6);

```

```
// Right headlight shadow map
glActiveTexture(GL_TEXTURE0 + 7);
glBindTexture(GL_TEXTURE_2D, rightHeadlightDepthMap);
glUniform1i(basicUniforms.rightHeadlightShadowMap, 7);
```

Shadow Calculations in the Main Fragment Shader During the final pass, the main fragment shader invokes the appropriate shadow function for each light. For example:

Listing 16: Shadow Calculations in Main Fragment Shader

```
float dirShadow = ShadowCalculation(normal,
    FragPosLightSpace, fPosition, DirectionalLightDir);
float pointShadow = PointShadowCalculation(normal,
    fPosition, PointLightDir);
float leftHeadlightShadow = HeadlightShadowCalculation(normal,
    FragPosLightSpaceLeftHeadlight, fPosition, LeftHeadlightDir,
    leftHeadlightShadowMap);
float rightHeadlightShadow = HeadlightShadowCalculation(normal,
    FragPosLightSpaceRightHeadlight, fPosition, RightHeadlightDir,
    rightHeadlightShadowMap);

// Combine each shadow factor with the light contribution
dirLightResult += (1.0 - dirShadow) *
     (diffuse * textureColor + specular * specularColor);
pointLightResult += (1.0 - pointShadow) *
     (diffuse * textureColor + specular * specularColor);
leftHeadlightResult += (1.0 - leftHeadlightShadow) *
     (diffuse * textureColor + specular * specularColor);
rightHeadlightResult += (1.0 - rightHeadlightShadow) *
     (diffuse * textureColor + specular * specularColor);
```

Here, a value of `1.0` for `dirShadow` indicates the fragment is entirely in shadow, so the light's diffuse and specular components are effectively reduced to zero. A shadow value of `0.0` means the fragment is fully lit by that light. Intermediate values soften or blend the result to achieve partial shadow penumbrae.

3.3.6 Dynamic Shadows

- The project updates `u_Time`, light positions, and object transformations each frame, causing the shadow maps to be regenerated when needed.
- Moving objects or lights automatically produce updated shadows for a more immersive, dynamic scene.

(a) Directional Light Shadows (b) Point Light Shadows (c) Spotlight Shadows

Figure 6: Shadow Mapping Comparisons: Shadows cast by directional, point, and spotlights.

Overall, these separate shadow passes (directional, point, and spotlight) collectively enhance the realism of the scene by casting accurate shadows from multiple light sources with different projections and volumes of influence.

3.4 Normal Mapping

Normal mapping is a method to simulate fine surface details on 3D models without increasing the actual geometric complexity. By using a normal map (an RGB texture encoding normal vectors in tangent space), we can modify per-pixel normals to create the illusion of bumps, grooves, and other high-frequency details.

3.4.1 Core Concept

- **Goal:** Enhance realism by adjusting each fragment's normal during lighting computations, making flat surfaces appear detailed.
- **Approach:** Store normals in a special texture (the *normal map*), and transform these from *tangent space* into *world space* (or view space) for proper lighting interactions.

3.4.2 Tangent & Bitangent Calculation

For correct normal mapping, each vertex must have a *tangent* and a *bitangent* in addition to its vertex normal. These three vectors (tangent, bitangent, normal) form a local orthonormal basis (the TBN matrix) that allows us to reinterpret normal-map data from texture (tangent) space into world (or view) space.

Implementation in `Model3D` Class

- **Compute Tangent and Bitangent per Triangle:**
  1. For each face with vertices `faceVertices[0..2]` and texture coordinates `faceTexCoords[0..2]`, calculate:
  2. Normalize the tangent and bitangent vectors.
  3. Assign them to each of the triangle's vertices.

Listing 17: Tangent and Bitangent Calculation Example

```
glm::vec3 edge1   = faceVertices[1] - faceVertices[0];
glm::vec3 edge2   = faceVertices[2] - faceVertices[0];
glm::vec2 deltaUV1 = faceTexCoords[1] - faceTexCoords[0];
glm::vec2 deltaUV2 = faceTexCoords[2] - faceTexCoords[0];

float f = 1.0f / (deltaUV1.x * deltaUV2.y - deltaUV2.x * deltaUV1.y);

glm::vec3 tangent, bitangent;

tangent.x   = f * (deltaUV2.y * edge1.x - deltaUV1.y * edge2.x);
tangent.y   = f * (deltaUV2.y * edge1.y - deltaUV1.y * edge2.y);
tangent.z   = f * (deltaUV2.y * edge1.z - deltaUV1.y * edge2.z);

bitangent.x = f * (-deltaUV2.x * edge1.x + deltaUV1.x * edge2.x);
bitangent.y = f * (-deltaUV2.x * edge1.y + deltaUV1.y * edge2.y);
bitangent.z = f * (-deltaUV2.x * edge1.z + deltaUV1.x * edge2.z);

tangent = glm::normalize(tangent);
bitangent = glm::normalize(bitangent);
```

- **Vertex Structure:** Each vertex holds:

Listing 18: Vertex Structure with Tangent and Bitangent

```
struct Vertex {
    glm::vec3 Position;
    glm::vec3 Normal;
    glm::vec2 TexCoords;
    glm::vec3 Tangent;
    glm::vec3 Bitangent;
};
```

3.4.3 TBN Matrix Construction in the Vertex Shader

Within the vertex shader, after transforming the vertex into world space:

1. **Apply Model Transform** to the tangent and bitangent:

   ```
   vec3 T = normalize(mat3(model) * vTangent);
   vec3 B = normalize(mat3(model) * vBitangent);
   vec3 N = normalize(mat3(model) * vNormal);
   ```

2. **Form the TBN Matrix**:

   ```
   TBN = mat3(T, B, N);
   ```

   This matrix is passed to the fragment shader for normal mapping.

Listing 19: TBN Matrix Construction in Vertex Shader

```
out mat3 TBN;

void main() {
    // ...
    vec3 T = normalize(mat3(model) * vTangent);
    vec3 B = normalize(mat3(model) * vBitangent);
    vec3 N = normalize(mat3(model) * vNormal);
    TBN = mat3(T, B, N);
    // ...
}
```

3.4.4 Using the Normal Map in the Fragment Shader

- **Sample Normal Texture:**

  ```
  vec3 normalMapColor = texture(normalTexture, fTexCoords).rgb;
  ```

- **Convert from 0–1 to -1–+1 Range:**

  ```
  normalMapColor = normalMapColor * 2.0 - 1.0;
  ```

  This maps the texture's [0..1] range to [-1..+1].

- **Transform to World Space (or View Space) via TBN:**

  ```
  vec3 normal = normalize(TBN * normalMapColor);
  ```

- **Use `normal` for Lighting:** Any subsequent lighting calculations (diffuse, specular, etc.) will use `normal` instead of the original vertex normal.

Listing 20: Key Fragment Shader Excerpt for Normal Mapping

```
if (useNormalMapping == 1) {
    vec3 sampledNormal = texture(normalTexture, fTexCoords).rgb * 2.0 - 1.0;
    normal = normalize(TBN * sampledNormal);
} else {
    normal = fNormal;
}
```

3.4.5 Texture Setup and Model Loading

- **Loading Normal Maps:**
  - In the model loader, detect if a material includes a *bump* or *normal* texture name (e.g., `bump_texname` in TinyObj).
  - Load it via `LoadTexture(..., "normalTexture")` and store it in the material's `textures` list.
- **Binding in the Render Pass:** In the final rendering pass, the normal map is typically bound to a specific texture unit (e.g., `GL_TEXTURE1`) and passed to the shader's `normalTexture` uniform.

3.4.6 Visual Impact of Normal Mapping

- **Enhanced Surface Detail:** Grooves, cracks, and ridges become visible when light interacts with the per-pixel normals, without extra vertices.
- **Low Performance Cost:** Normal mapping uses a texture fetch plus a matrix multiply per fragment, which is typically cheaper than adding additional geometric detail.
- **Comparison Examples:** Figures 7a–8d show the stark difference between rendering with and without normal mapping.

(a) Without Normal Mapping (b) With Normal Mapping (c) Without Normal Mapping (d) With Normal Mapping

Figure 7: Normal Mapping Comparisons: Surface details with and without normal mapping (Comparisons 1 and 2).

(a) Without Normal Mapping (b) With Normal Mapping (c) Without Normal Mapping (d) With Normal Mapping

Figure 8: Normal Mapping Comparisons: Surface details with and without normal mapping (Comparisons 3 and 4).

In summary, the normal mapping pipeline is:

1. **Precompute tangents and bitangents** for each vertex in the model loader.
2. **Store tangent/bitangent** in the vertex structure and upload them as attributes.
3. In the **vertex shader**, build the TBN matrix from `tangent`, `bitangent`, and `normal`.
4. In the **fragment shader**, **sample the normal map**, convert from [0..1] to [-1..+1], then transform it by the TBN.
5. **Use the transformed normal** in lighting calculations to achieve detailed surface shading.

This approach ensures that even low-polygon objects can exhibit complex lighting responses, enhancing the overall visual fidelity of the scene.

3.5 Effects

The project incorporates advanced effects to enhance realism and immersion in the 3D forest scene. These include a dynamic rain simulation and other integrated lighting features. This section details how the rain effect is initialized, updated, and rendered.

3.5.1 Rain Simulation

Overview The rain simulation system generates and animates a large number of raindrops falling through the scene. It is designed to be both performant (handling up to one million raindrops) and visually convincing, integrating with the existing lighting pipeline (directional, point, and spotlights).

- **Number of raindrops:** `NUM_RAINDROPS = 1000000`
- **Geometry:** Each raindrop is rendered as a *point* primitive expanded in a *geometry shader* into a small quad or trapezoid stretched downwards.
- **Lighting:** Uses diffuse and specular computations (Blinn-Phong) for each raindrop, sampling scene lights.
- **Reflections:** A cubemap (`environmentMap`) is sampled to create subtle reflections on the raindrops.

Initialization

- **Data Structures:**
  1. `raindrops`: A `std::vector<glm::vec3>` storing the positions of each raindrop.
  2. `raindropsVelocity`: A `std::vector<glm::vec3>` storing the velocity (mostly in the negative y-direction).
  3. `raindropParams`: A `std::vector<glm::vec2>` containing per-drop parameters:
     - `x` = length factor
     - `y` = speed factor
- **Random Generation:**
  - Each raindrop is assigned a random (x, z) within `[rainMinX, rainMaxX]` and `[rainMinZ, rainMaxZ]`, and a random `y` position near `rainTopY`.
  - `lengthFactor` and `speedFactor` are random scalars controlling the drop's appearance and velocity.
- **GPU Buffers:**
  - `rainVAO`, `rainVBO`: Store the (x, y, z) positions of the raindrops, updated every frame.
  - `rainParamsVBO`: Stores the (`lengthFactor`, `speedFactor`) per drop. This remains mostly static.
  - Vertex attribute layout:

    ```
    // layout (location = 0) -> vec3 inPosition
    // layout (location = 1) -> vec2 inParams
    ```

- **Gravity and Terminal Velocity:**

  ```
  #define GRAVITY -4.905f
  #define TERMINAL_VELOCITY -25.0f
  ```

  Ensures drops accelerate downward but do not exceed a certain speed.

Update Mechanism

- **Per-frame Updates:**
  1. For each raindrop, add `GRAVITY` * `speedFactor` * `deltaTime` to its vertical velocity if it has not reached `TERMINAL_VELOCITY`.
  2. Update position: `raindrops[i] += raindropsVelocity[i] * deltaTime`.
  3. If a raindrop falls below `rainBottomY`, reinitialize it at the top (`rainTopY`) with new random parameters.
- **Buffer Update:**

Listing 21: GPU Buffer Update for Raindrop Positions

```
    glBindBuffer(GL_ARRAY_BUFFER, rainVBO);
    glBufferSubData(GL_ARRAY_BUFFER, 0,
      NUM_RAINDROPS * sizeof(glm::vec3),
      &raindrops[0]);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
```

This updates the GPU with the new positions each frame.

Rendering Pipeline

1. **Vertex Shader (`rain.vert`):**

Listing 22: Vertex Shader (rain.vert)

```
#version 410 core

layout (location = 0) in vec3 inPosition;
layout (location = 1) in vec2 inParams;

uniform mat4 view;
uniform mat4 projection;

out vec3 vPosition;
out float vLengthFactor;
out float vSpeedFactor;

void main() {
    vPosition = inPosition;
    vLengthFactor = inParams.x;
    vSpeedFactor = inParams.y;
    gl_Position = projection * view * vec4(inPosition, 1.0);
}
```

- Passes the position and rain-specific parameters (`lengthFactor`, `speedFactor`) to the next stage.

2. **Geometry Shader (`rain.geom`):**

```
layout(points) in;
layout(triangle_strip, max_vertices = 4) out;

// inPosition -> turned into a short "streak"
```

- Takes a single point and emits up to 4 vertices forming a small trapezoid or strip. - The length in the y-direction depends on `baseDropLength` $\cdot$ `vLengthFactor`. - Includes a slight curvature offset, thickness, etc.

3. **Fragment Shader (`rain.frag`):**

Listing 23: Fragment Shader (rain.frag) Snippet

```
#version 410 core
...
uniform vec4 rainColor;
uniform samplerCube environmentMap;
...
// 1) Compute alpha via motion blur intensity, drop position along the
    streak.
// 2) Compute lighting from directional/point/spot/headlights.
// 3) Optional reflection sampling from the environment map for subtle wet
    shine.
// 4) Combine final color = lighting * rainColor + reflection.
```

- Also uses Blinn-Phong model, sampling the same light structures as the rest of the scene. - Applies alpha blending for transparency, especially toward the top and bottom of the streak.

Lighting Integration

- The same directional, point, and spotlight structs are passed into the rain fragment shader.
- Rain drops receive standard diffuse + specular calculations.
- **Reflection:** A small `environmentMap` lookup (`reflect(viewDir, norm)`) adds the effect of raindrops reflecting the skybox.
- **Specular in Main Scene:**
  - When `rainEnabled` is set in the main fragment shader, Blinn-Phong specular is calculated for the wet surfaces using `specularColor` from a specular map.
  - Each light's specular function (e.g., `computeSpecularDirLight`) returns a value that's multiplied by `specularColor`, further simulating the shiny effect of wet surfaces.

Final Rendering Steps

- **Enable Blending:**

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

- **Draw Rain as Points:**

```
glBindVertexArray(rainVAO);
glDrawArrays(GL_POINTS, 0, NUM_RAINDROPS);
glBindVertexArray(0);
```

- **Disable Blending:**

```
glDisable(GL_BLEND);
```

Main Loop Integration

- `updateRain(deltaTime)` is called each frame to update positions and velocities.
- `renderRain()` is called if `rainEnabled` is `true`.
- The scene's camera matrices (`view` and `projection`) are uploaded to the rain shader, along with the current `environmentMap`.

Summary By combining a point-based geometric approach with a sophisticated geometry shader, the system efficiently handles up to a million raindrops. The Blinn-Phong lighting calculations and environment reflections seamlessly blend the rain with the rest of the scene's lighting. When `rainEnabled` is active, surfaces in the *main fragment shader* can also switch to a "wet" specular model (using the loaded specular texture), intensifying highlights and adding to the immersive, rainy atmosphere.

3.5.2 Fog Effect

Fog is implemented as a dynamic environmental effect that enhances the scene's depth and mood by obscuring distant objects and gradually blending them with a fog color. This system supports two primary modes: *Layered Fog* and *Animated Fog*, each using different techniques for density calculation. The resulting fog factor is used to blend the object's color with the fog color, creating a more immersive and realistic atmosphere.

Overview

- **Fog Modes:**
  - **Layered Fog:**
    - Varies in density as a function of height (or altitude).
    - Simulates denser fog at lower altitudes and clearer air at higher altitudes.
    - Requires a fast, layer-based integral to compute overall density between the camera and the fragment.
  - **Animated Fog:**
    - Introduces time-based noise functions for swirling, wind-like motion in the fog.
    - More dynamic, creating a sense of rolling or drifting mist throughout the scene.
- **Time-of-Day and Weather Adaptation:**
  - Fog color, density, and distance parameters interpolate based on day/night cycles or other conditions.
  - During daytime, the fog end distance (`gFogEnd`) is larger with a lighter color; at night, it is darker, denser, and closer.

Shader Integration Fog is applied in the *fragment shader* by mixing the computed fog color with the fragment's final color. The key uniforms and functions are:

- **Uniforms:**

Listing 24: Fog Uniforms

```
uniform int fogEnabled;
uniform vec3 fogColor;
uniform float gFogEnd;
uniform float gLayeredFogTop;
uniform float gFogTime;
uniform float gExpFogDensity;
uniform vec3 viewPos; // Camera position
...
```

- **Fog Factor Computation:**

1. **Layered Fog** (`CalcLayeredFogFactor()`):

Listing 25: Layered Fog Factor Calculation

```
float CalcLayeredFogFactor()
{
    // 1) Project camera and fragment positions onto ground plane
    //    to compute horizontal distance ratio (DeltaD).
    // 2) Compute height differences and integrate a fog-density
        function.
    // 3) Return exp(-FogDensity), which decreases with greater
        distance
    //    and height difference.
}
```

2. **Animated Fog (`CalcAnimatedFogFactor()`):**

Listing 26: Animated Fog Factor Calculation

```
float CalcAnimatedFogFactor()
{
    // 1) Base factor = exp(-DistRatio * gExpFogDensity).
    // 2) AnimFactor = trigonometric noise using (fPosition, gFogTime).
    // 3) FogFactor = baseExp + DistRatio * AnimFactor.
}
```

3. **Final Mixing in the Fragment Shader:**

Listing 27: Fragment Shader Fog Mixing

```
if (fogEnabled == 1) {
    float fogFactor = 0.0;
    if (windEnabled == 1) {
        // If wind is on, we use Animated Fog
        fogFactor = CalcAnimatedFogFactor();
    } else {
        // Otherwise, use Layered Fog
        fogFactor = CalcLayeredFogFactor();
    }
    fogFactor = clamp(fogFactor, 0.0, 1.0);

    finalResult = mix(fogColor, finalResult, fogFactor);
}
```

- A `fogFactor` of `0.0` means *fully fogged* (scene color replaced by `fogColor`), and `1.0` means *no fog* (fragment color unchanged).

Implementation Details

Layered Fog Computation

- The code projects both the camera position (`viewPos`) and the fragment position (`fPosition`) onto the ground plane (zeroing out the y component).
- Calculates the horizontal distance ratio ΔD as $\frac{\|\texttt{CameraProj} - \texttt{PixelProj}\|}{\texttt{gFogEnd}}$.
- Determines how much the fog density accumulates from the camera's y-height to the fragment's y-height:
  - If the camera is above `gLayeredFogTop` and the fragment is below it, a vertical slice is computed.
  - Various intermediate steps measure the fraction of the path that dips into dense fog near the ground.
- The final exponent is taken of this integrated density to produce a smooth falloff.

Animated Fog Computation

- Evaluates an exponential distance-based term: $\exp(-\texttt{DistRatio} \times \texttt{gExpFogDensity})$.
- Adds trigonometric noise terms that shift with `gFogTime`, making the fog move or swirl:

```
float AnimFactor = -(1.0
  + 0.5 * cos(5.0 * PI * z + gFogTime)
  + 0.2 * cos(7.0 * PI * (z + 0.1*x))
  + ... );
```

- Produces a final `FogFactor` by blending these noise components with the base exponential factor.

Main-Class Control

- **Toggling Fog:** A boolean `fogEnabled` is used in the shader to activate or deactivate the effect.
- **Time-of-Day Blending:**

Listing 28: Fog Color and Parameters Interpolation

```
glm::vec3 currentFogColor = mix(nightFogColor, dayFogColor, blend);
float currentgFogEnd = mix(nightgFogEnd, daygFogEnd, blend);
float currentgExpFogDensity = mix(nightgExpFogDensity, daygExpFogDensity,
    blend);
float currentgLayeredFogTop = mix(nightgLayeredFogTop, daygLayeredFogTop,
    blend);
```

- **Fog Time Advancement:** `fogTime` $+= 0.00075f \times (\dots) \times$ `deltaTime` drives the animated fog's temporal variation.

Visual Impact

- Distant objects gradually fade into fog color, giving depth cues and a sense of scale.
- Layered fog emphasizes vertical variation, such as valley mist or cloud layers.
- Animated fog adds swirling, drifting effects that respond to wind or time changes, producing a lively atmosphere.

Final Rendering After lighting, shadows, and other effects (such as normal mapping or rain) are computed in the fragment shader, the fog step is applied last:

```
finalResult = mix(fogColor, finalResult, fogFactor);
```

This ensures all shading results are correctly integrated into the fog color. By adjusting `fogEnabled`, `gFogEnd`, `gLayeredFogTop`, and `gExpFogDensity`, you can seamlessly switch between clearer or denser atmospheres, layered or animated appearances, or even turn fog off entirely.

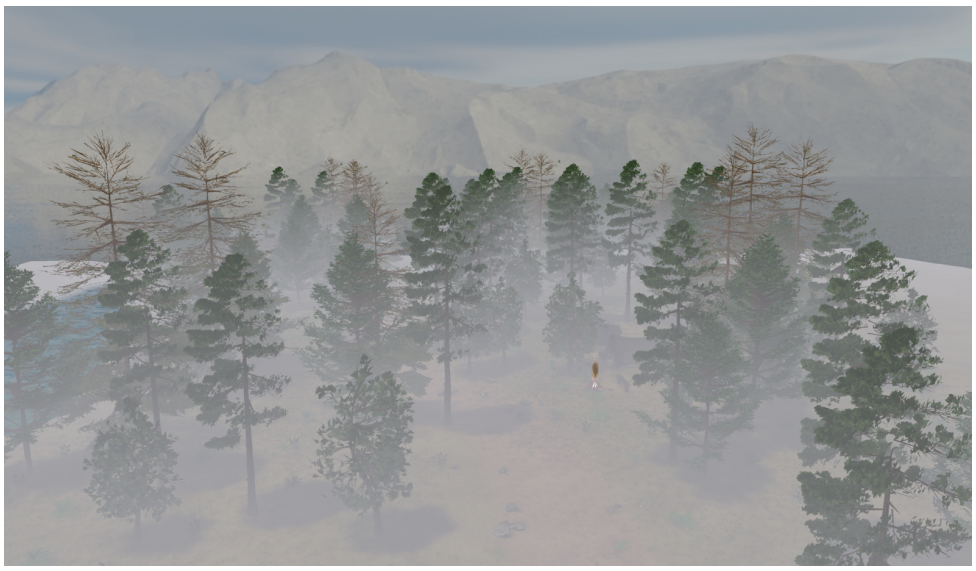


Figure 9: Layered Fog Effect

Figure 10: Animated Fog Effect

3.5.3 Wind Effect

The wind effect introduces dynamic motion to vegetation and other wind-responsive elements (e.g., grass, leaves, and ferns). By combining Perlin noise, sine waves, and object-type–based multipliers, the scene gains a realistic sense of natural movement. The following sections explain how wind parameters are set, how the shaders apply the effect, and how different objects respond to varying wind intensities.

Overview

- **Wind Parameters:**
  - `windEnabled`: Toggles the wind effect on or off.
  - `u_Time`: Tracks global time to animate wind dynamics.
  - `u_WindDirection`: Specifies the (normalized) direction of the wind.
  - `u_WindStrength`: Scales the overall intensity of wind displacement.
  - `gustSize`: Controls the spatial scale of wind gust noise.
  - `gustSpeed`: Controls how quickly wind gust noise evolves over time.
  - `windWaveLength`: Governs the sine wave's periodicity for large- and medium-scale motion.
- **Object-Specific Wind Types:**
  - **Grass / Stems** (`objectType = 0`): Show moderate motion, influenced by height.
  - **Ferns** (`objectType = 2`): Medium responsiveness, swaying somewhat more visibly.
  - **Leaves / Default** (`objectType = 1` or anything else): Generally the highest wind response.
- **Material Flag:**
  - `isWindMovable`: Determines if a particular batch of geometry is affected by wind.
  - Set during model loading by examining material names (e.g., "Leaf", "Grass", or "Fern").

Wind Dynamics in the Vertex Shader When `windEnabled` and `isWindMovable` are both true, vertex positions are displaced based on:

1. **Height-based multiplier:**

   ```
   float getWindMultiplier(int objectType, float height,
                           float minHeight, float maxHeight);
   ```

   This function produces larger displacement for taller objects (e.g., tree canopies) and smaller displacement for grasses, depending on `objectType` and the vertex's `y` position.

2. **Perlin 3D Noise:**

   ```
   float noise = Perlin3DNoise(vec3(
       pos.x / u_GustSize,
       pos.z / u_GustSize,
       u_Time * u_GustSpeed
   ));
   ```

   The noise value modulates the amplitude of wind effects, ensuring random variation across space and time.

3. **Layered Sine Waves:**

   ```
   // Large Wind Power
   float largeWindPower = sin(windFactor) * windMultiplier;
   ...
   pos.x += largeWindPower * windDir.x;
   pos.z += largeWindPower * windDir.z;
   ```

   - **Large-scale motion**: Uses a sine wave of relatively low frequency to simulate big gusts.
   - **Medium-scale motion**: Adds a second sine wave with a different frequency/amplitude for mid-frequency variations.
   - **Small-scale motion**: High-frequency jitter along the vertex normal, simulating leaf flutter or grass blade trembling.

Key Code Snippet:

Listing 29: Wind Movement Conditional

```
if (windEnabled == 1 && isWindMovable == 1) {
    float windMultiplier = getWindMultiplier(u_ObjectType, vPos.y,
                                             -1.0, 1.0);
    vec3 windDir = normalize(u_WindDirection) * windMultiplier;

    float windFactor = (pos.x + pos.y + pos.z) /
                       u_WindWaveLength + u_Time;

    float noise = Perlin3DNoise(vec3(
        pos.x / u_GustSize,
        pos.z / u_GustSize,
        u_Time * u_GustSpeed
    ));

    // Large Wind Power
    float largeWindPower = sin(windFactor) * windMultiplier;
    // apply partial damping depending on sign
    if (largeWindPower < 0.0) largeWindPower *= 0.4;
    else                      largeWindPower *= 0.6;
    largeWindPower *= noise;
    pos.x += largeWindPower * windDir.x;
```

```
    pos.z += largeWindPower * windDir.z;

    // Medium Wind Power
    float x = (2.0 * sin(1.0 * windFactor)) + 1.0;
    float z = (1.0 * sin(1.8 * windFactor)) + 0.5;
    vec3 mediumWindPower = vec3(x, 0.0, z) *
                           vec3(0.1) * noise * windMultiplier;
    pos += mediumWindPower;

    // Small Wind Power (jitter along normal)
    float smallWindPower = 0.065 * sin(2.650 * windFactor);
    smallWindPower *= u_WindStrength * windMultiplier;

    vec3 smallJitter = vec3(smallWindPower) * vNormal *
                       vec3(1.0, 0.35, 1.0) * 0.075 * noise;
    pos += smallJitter;
}
```

Wind Direction Updates in the Main Loop

- **Time-based Rotation:**

  ```
  float windAngle = u_Time * 0.15f;
  u_WindDirection = glm::normalize(
      glm::vec3(cos(windAngle), 0.0f, sin(windAngle))
  );
  ```

  This ensures the wind direction slowly rotates with time, simulating gusts shifting around in a broad circle.

- **Uniform Upload:**

  ```
  glUniform3fv(basicUniforms.u_WindDirection, 1,
               glm::value_ptr(u_WindDirection));
  ```

Integration with Shadow Mapping Because the same vertex transformations (wind displacements) occur in both the `main` and the `shadow` vertex shaders, moving vegetation will cast dynamically updated shadows:

- **Shadow Shaders:** Each light's shadow-vertex shader repeats the Perlin noise and sine wave logic, ensuring the geometry's displaced position matches.
- **Consistent Shadows:** Vegetation motion is reflected in the depth maps, so shadows follow the swaying plants accurately.

Object Type Management

- **Model Loading:**
  - If a material name contains "Grass", "Leaf", or "Fern", `isWindMaterial` is set to true.
  - Stored in `meshbatch.isWindMovable`; a uniform `isWindMovable = 1` is then passed to the shader.
- **Object Type Uniform:**

  ```
  if (objectTypeLoc != -1) {
      glUniform1i(objectTypeLoc,
         (isGrass ? 0 :
           (isFern  ? 2 :
                      1)));
  ```

```
}
if (isWindMovableLoc != -1) {
    glUniform1i(isWindMovableLoc,
                isWindMovable ? 1 : 0);
}
```

Summary of the Wind Pipeline

1. **Enable Wind:** Set `windEnabled` = `true`.
2. **Compute Wind Direction:** Slightly rotate `u_WindDirection` each frame using `u_Time`.
3. **Upload Uniforms:** `u_WindStrength`, `gustSize`, `gustSpeed`, `windWaveLength`, ...
4. **In Vertex Shaders**:
   - Check (`windEnabled` $\wedge$ `isWindMovable`).
   - Displace each vertex with layered sine waves and Perlin noise.
   - Adjust motion based on `u_ObjectType`, `heightFactor`, and `u_WindStrength`.
5. **Shadow Consistency**: The same transformation logic is repeated in shadow vertex shaders, ensuring animated shadows.

This approach yields a responsive, natural wind effect: grass ripples, leaves flutter, and larger plants sway in the breeze. Both geometry and shadows remain synchronized, blending seamlessly with other scene effects (fog, rain, normal mapping) to enhance realism and immersion.

3.5.4 Fire Effect

The fire effect creates a visually dynamic campfire environment by simulating flames, smoke, and ember particles. This system employs a GPU-accelerated particle approach, diverse textures in a 2D array, and multiple update passes to capture the fluid nature of fire. Below is a detailed breakdown for replicating the effect.

Overview of the Fire System

- **Particle System Architecture:**
  - The effect uses instanced rendering for thousands of fire-related particles.
  - Particles are categorized into three types:
    1. **Flame Particles** (Type `0`): Represent the bright, flickering core of the fire.
    2. **Smoke Particles** (Type `1`): Darker, slowly rising plumes.
    3. **Ember Particles** (Type `2`): Small glowing sparks that occasionally trail off.
- **Texture Array:**
  - A 2D texture array (`fireTextureArray`) stores multiple flame textures.
  - Each particle references a layer index (`inTexIndex`), sampling a different slice for variety.
- **Draw Passes:**
  - **Pass 1**: Renders flame and ember particles with additive blending.
  - **Pass 2**: Renders smoke particles with `SRC_ALPHA` and `ONE_MINUS_SRC_ALPHA` for semi-translucent smoke.

Initialization and Buffer Setup

- **Particle Structures:**

Listing 30: FireParticle Structure

```
struct FireParticle {
    glm::vec3 position;
    glm::vec4 color;
    float size;
    float rotation;
    float rotationSpeed;
    float life;
    float initialLife;
    glm::vec3 velocity;
    int type;          // 0=flame, 1=smoke, 2=embers
    int textureIndex;  // which layer to sample in fireTextureArray
};
```

- **Texture Array Loading:**

  1. Collect file paths for multiple fire images (e.g., `fire1.png`, `fire2.png`, ...).
  2. Call `LoadFireTextureArray(...)` to create an `GL_TEXTURE_2D_ARRAY` with each image as a separate layer.

- **Particle Spawning:**

  - `respawnFireParticle()`, `respawnSmokeParticle()`, `respawnEmberParticle()` assign random positions around the campfire, random velocities, lifetimes, and colors.
  - `MAX_FIRE_PARTICLES`, `MAX_SMOKE_PARTICLES`, `MAX_EMBER_PARTICLES` define spawn counts.
  - Each type is given unique velocity ranges (e.g., flames move up faster, smoke drifts slowly upward, embers can fly off).

- **GPU Buffers:**

  - `quad2Vertices`: A small 2D quad used as the *billboard* for each particle.
  - `quad2VAO`, `quad2VBO`: Store the static quad.
  - `instanceVBO`: Stores `FireParticle` data (`position`, `color`, `size`, `rotation`, `textureIndex`, ...) for instancing.

Update Mechanism

- **Time-based Dynamics:**

Listing 31: Update Fire Function

```
void updateFire(float deltaTime, float globalTime) {
    for (auto& p : fireParticles) {
        p.life -= deltaTime;
        if (p.life <= 0.0f) {
            // Respawn as flame, smoke, or ember
        }
        // ...
        p.position += p.velocity * deltaTime;
    }
    // Upload updated data to instanceVBO
}
```

- **Particle Lifecycle:**

  - `p.life` counts down each frame; when it expires, the particle is respawned (often with a small random chance to switch types).
  - `lifeRatio = p.life / p.initialLife` is used to gradually fade color, size, and alpha.

- **Velocity Adjustments:**
  - **Swirl Effects:** Apply a small rotation to $\langle v_x, v_z \rangle$ each frame.
  - **Perlin Noise:** A noise vector is added to `p.velocity` for random drift.
  - **Type-Specific Behavior:**
    * **Flames (0)**: Rise quickly, flicker in color, clamp at a max height, and occasionally surge upward.
    * **Smoke (1)**: Move more slowly upward, with a larger size and a more transparent fade.
    * **Embers (2)**: Can spawn trailing embers, have variable speed, and fade out faster.

Listing 32: Fire Vertex Shader

```
#version 410 core

layout (location = 0) in vec2 inQuadPos;
layout (location = 1) in vec2 inTexCoord;
layout (location = 2) in vec3 inPosition;
layout (location = 3) in vec4 inColor;
layout (location = 4) in float inSize;
layout (location = 5) in float inRotation;
layout (location = 6) in int   inTexIndex;

// Outputs
out vec2 TexCoord;
out vec4 ParticleColor;
flat out int texLayer;

// Uniforms
uniform mat4 view;
uniform mat4 projection;
uniform vec3 cameraRight;
uniform vec3 cameraUp;
uniform float flameAspectX;
uniform float flameAspectY;
uniform float time;

void main()
{
    float rad  = radians(inRotation);
    float cosT = cos(rad);
    float sinT = sin(rad);
    // rotate billboard corners
    vec2 rotatedPos = vec2(
        inQuadPos.x * cosT - inQuadPos.y * sinT,
        inQuadPos.x * sinT + inQuadPos.y * cosT
    );

    // small swirl near top of flame
    if (inQuadPos.y > 0.3) {
        float swirl = sin((inQuadPos.x + time * 3.0) * 10.0) * 0.03;
        rotatedPos.x += swirl * (inQuadPos.y - 0.3) * 1.5;
    }

    // height-based taper
    float heightFactor = clamp((inQuadPos.y + 0.5) / 1.0, 0.0, 1.0);
```

```
    float taper = mix(1.0, 0.75, heightFactor);
    vec2 taperedPos = rotatedPos * vec2(taper, 1.0);

    // further distortion
    float distortion = sin(taperedPos.x * 12.0 + time * 6.0) * 0.02;
    taperedPos.y += distortion;

    // billboard in world space
    vec3 worldPos = inPosition
        + cameraRight * (taperedPos.x * inSize * flameAspectX)
        + cameraUp    * (taperedPos.y * inSize * flameAspectY);

    gl_Position   = projection * view * vec4(worldPos, 1.0);
    TexCoord      = inTexCoord;
    ParticleColor = inColor;
    texLayer      = inTexIndex;
}
```

Key points:

- Particles always face the camera (*billboarding*) via `cameraRight` and `cameraUp`.
- Random swirl and distortion for a lively flame shape.
- `flameAspectX` and `flameAspectY` can stretch the billboard (e.g., a taller flame).

Listing 33: Fire Fragment Shader

```
#version 410 core

in vec2 TexCoord;
in vec4 ParticleColor;
flat in int texLayer;
out vec4 FragColor;

uniform sampler2DArray fireTextureArray;

void main()
{
    // sample from the array layer
    vec3 arrayCoord = vec3(TexCoord, float(texLayer));
    vec4 texSample = texture(fireTextureArray, arrayCoord);

    // radial fade
    float dist = length(vec2((TexCoord.x - 0.5) * 0.8,
                             TexCoord.y - 0.5));
    float radialFade = 1.0 - smoothstep(0.30, 0.45, dist);

    // top fade
    float topFade = smoothstep(0.70, 1.00, TexCoord.y);

    // combine
    float combinedFade = radialFade * (1.0 - topFade);
    vec4 outColor = ParticleColor * texSample * combinedFade;

    // subtle glow
    float glowAmount = outColor.a * 0.07;
    outColor.rgb += glowAmount;

    // discard nearly invisible
    if (outColor.a < 0.02)
        discard;

    FragColor = outColor;
}
```

Key points:

- Blends sampled texture color with `ParticleColor`, further modulated by radial and top fades.
- Produces a soft edge at the particle boundary and near the top.
- A small "glow" factor brightens the result.

Rendering Pipeline

1. **Update Fire:** (`updateFire()`)
   - Decrement particle lifetimes; respawn if expired.
   - Adjust velocity with swirl, noise, or type-specific logic.
   - For instance, flames get a random color flicker, smoke rises more slowly.
   - Upload changes to `instanceVBO`.
2. **Render Fire:** (`renderFire()`)

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE); // additive
// Draw flame & ember pass
...

glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA); // alpha blending
// Draw smoke pass
```

3. **Layered Blending:**
   - Flames + embers benefit from additive blending to appear bright and emissive.
   - Smoke is semi-opaque, so it uses alpha blending.
4. **Depth Mask and Sorting:**
   - Often, depth writes are disabled (`glDepthMask(GL_FALSE)`) when drawing particles.
   - Sorting by distance can further improve rendering, but this example draws them in two passes: bright stuff first, then smoke.

Flickering Integration with Scene Lights

- **Light Flicker:**
  - The campfire's point light (`pointLight`) can vary in intensity:

```
float flicker = 1.0
  + sin(globalTime * FLICKER_BASE_FREQUENCY)
    * FLICKER_AMPLITUDE_AMBIENT
  + glm::linearRand(-FLICKER_RANDOMNESS, FLICKER_RANDOMNESS);

pointLight.ambient  *= flicker;
pointLight.diffuse  *= (1.0 + flicker * FLICKER_AMPLITUDE_DIFFUSE);
pointLight.specular *= (1.0 + flicker * FLICKER_AMPLITUDE_SPECULAR);
```

  - This ensures lighting from the fire also "breathes" along with the flames, adding to realism.

Putting It All Together (Main Loop)

- **1)** Call `updateFire(deltaTime, currentTime)` each frame.
- **2)** Render other scene objects (e.g., terrain, models).
- **3)** If `pointLight.enabled`, call `renderFire(currentTime)`.
- **4)** Optionally update `pointLight` intensities for flicker.

Summary This fire system uses GPU-instanced particles with layered textures, Perlin noise, swirl effects, and multiple fade functions to mimic the swirling, flickering nature of a campfire. By separating flames, smoke, and embers, each with its own update logic and blending pass, the effect combines to create a convincing, dynamic fire that integrates seamlessly into the scene's overall lighting and atmosphere.

3.5.5 Lightning Effect

The lightning effect introduces momentary flashes of intense light during rainstorms, heightening the scene's atmosphere and drama. Below is an in-depth description of how this effect is set up and controlled in code, including timing, pulsing, and integration with the global lighting model.

Implementation Overview

- **Randomized Timing:**
  - After a random interval between `flashIntervalMin` and `flashIntervalMax`, a lightning flash is triggered.
  - Each flash lasts a short random duration (`flashDuration`).
  - When a flash ends, a new random interval is selected for the *next* flash.
- **Flashlight Integration:**
  - A dedicated light source (`flashLight`) is enabled during the flash, then disabled after the flash ends.
  - `flashLight.direction`, `flashLight.color`, and other properties are randomized to simulate different strike angles and intensities.
  - The `globalLightIntensity` is temporarily raised to brighten the entire scene.
- **Lighting Pulses:**
  - During a flash, multiple pulses can occur at random intervals (between `pulseIntervalMin` and `pulseIntervalMax`).
  - Each pulse adds a burst of brightness (e.g., `intensity`) for a short duration, simulating flickering lightning illumination.
  - Pulses accumulate in `activePulses`, then decay over time, with `globalLightIntensity` clamped to prevent overexposure.

Listing 34: Core Variables and Logic

```
1    // Lightning Booleans & Timers
2    bool isFlashing = false;
3    float flashDuration = 0.2f;
4    float flashTimer = 0.0f;
5    float globalLightIntensity = 1.0f;
6    float flashIntervalMin = 7.0f;
7    float flashIntervalMax = 14.0f;
8    float nextFlashTime = 0.0f;
```

```
    float pulseIntervalMin = 0.05f;
    float pulseIntervalMax = 0.3f;
    float timeSinceLastPulse = 0.0f;

    struct IntensityPulse {
        float duration;
        float timer;
        float intensity;
    };
    std::vector<IntensityPulse> activePulses;

    ...

    // Randomly assign the initial nextFlashTime:
    nextFlashTime = static_cast<float>(rand()) / RAND_MAX
                    * (flashIntervalMax - flashIntervalMin)
                    + flashIntervalMin;
```

Main Loop Integration if `rainEnabled` is true:

Listing 35: Wind Flash Handling

```
if (!isFlashing) {
    // Countdown to next flash
    nextFlashTime -= deltaTime;
    if (nextFlashTime <= 0.0f) {
        // Start a new flash
        isFlashing = true;
        flashDuration = 0.3f + ... // random [0.3..1.5]
        flashTimer = flashDuration;

        // Reset pulses, enable flashLight, randomize direction/color
        ...
        globalLightIntensity = 1.0f;
        ...
        // Also schedule thunder (sound) after a delay
        isThunderScheduled = true;
        thunderDelay = 0.8f + ...
        thunderTimer = thunderDelay;
    }
} else {
    // Flash is active
    flashTimer -= deltaTime;
    if (flashTimer <= 0.0f) {
        // End flash
        isFlashing = false;
        flashLight.enabled = false;
        globalLightIntensity = 1.0f;
        ...
        activePulses.clear();
        timeSinceLastPulse = 0.0f;
        // Next flash
        nextFlashTime = ... // random interval again
    }

    // Check for pulses
    timeSinceLastPulse += deltaTime;
    float randPulseInterval = ... // random [pulseIntervalMin..pulseIntervalMax]
    if (timeSinceLastPulse >= randPulseInterval) {
        float pulseIntensity = 2.0f + ...
        float pulseDuration = 0.05f + ...
        activePulses.push_back(IntensityPulse{ pulseDuration, pulseDuration,
            pulseIntensity });
```

```
        timeSinceLastPulse = 0.0f;
    }

    // Update global intensity from pulses
    updateGlobalLightIntensity(deltaTime);
}
```

Listing 36: Updating Global Light Intensity with Pulses

```
void updateGlobalLightIntensity(float deltaTime) {
    // Decay existing pulses
    for (auto it = activePulses.begin(); it != activePulses.end(); ) {
        it->timer -= deltaTime;
        if (it->timer <= 0.0f) {
            it = activePulses.erase(it);
        } else {
            ++it;
        }
    }

    // Base intensity is 1.0f
    globalLightIntensity = 1.0f;
    // Accumulate from active pulses
    for (const auto& pulse : activePulses) {
        globalLightIntensity += pulse.intensity;
    }

    // Clamp to prevent overexposure
    globalLightIntensity = glm::clamp(globalLightIntensity, 1.0f, 15.0f);

    // Update shader
    myBasicShader.useShaderProgram();
    glUniform1f(basicUniforms.globalLightIntensity, globalLightIntensity);
}
```

Shader Integration (Fragment Shader) In the main fragment shader, we apply:

1. The `globalLightIntensity` multiplier to the final color.
2. A bluish tint if `flashLight.enabled` is true.
3. Additional diffuse/specular from the flashlight if it's enabled and oriented like lightning.

Listing 37: Flashlight Contributions in Fragment Shader

```
    // Compute flashlight contributions
    if (flashLight.enabled == 1) {
        vec3 ambient  = flashLight.ambient * flashLight.color;
        vec3 diffuse  = computeDiffuseDirLight(normal,
                                normalize(-flashLight.direction));
        vec3 specular = vec3(0.0);

        if(useBlinnPhong == 1) {
            specular = computeSpecularDirLight(normal, viewDir,
                                        normalize(-flashLight.direction));
        }

        vec3 flashLightResult = ambient * textureColor;
        flashLightResult += diffuse * textureColor + specular * specularColor;
        finalResult += flashLightResult;
    }

    finalResult *= globalLightIntensity;
```

```
19
20      if (flashLight.enabled == 1) {
21          vec3 flashTint = vec3(0.5, 0.5, 1.0);
22          float tintStrength = 0.3;
23          finalResult = mix(finalResult, flashTint * finalResult, tintStrength);
24      }
```

Visual Impact

- The brief but intense flashes illuminate the scene, revealing detail otherwise hidden in darkness or rain.
- Pulsing intensities simulate flickers of real lightning, avoiding a static, uniform flash.
- Combined with *fog* and *rain*, lightning accentuates the stormy atmosphere, making the forest feel more dramatic.

Summary The lightning effect is powered by a randomized timer for starting flashes, a short active window for illumination, and quick pulses to simulate flickers. During a flash, a special flashlight source is enabled, raising the entire scene's brightness and applying a bluish tint to the final color. After the flash ends, intensity resets to normal, and a new random interval begins. Together, these elements create dynamic, unpredictable lightning that enhances immersion during rainstorms.

3.5.6 Audio Effects

Audio effects significantly enhance the immersive experience of the forest scene by providing dynamic environmental sounds synchronized with visual events (rain, thunder, fire). The audio pipeline is built using the `miniaudio` library, enabling multiple spatialized or non-spatialized sound sources to run concurrently.

Overview of Audio Components

- **Rain Sound:**
  - A looping rain sound is played when the `rainEnabled` flag is active.
  - Volume can be controlled via keyboard inputs (e.g., `F2`/`F3`) or programmatically (e.g., `audioMgr->increaseRai`
  - The position of the rain sound can be updated if desired (though typically it is a global ambient sound, so spatialization may be disabled).
- **Thunder Sound:**
  - Triggered during lightning events to create realism (via `playThunder()`).
  - A random delay is introduced between the lightning flash and the thunder playback (`thunderTimer`), imitating real-world timing.
  - Volume is adjustable. In this implementation, it is typically non-spatial or only moderately spatial to mimic distant thunder.
- **Fire Sound:**
  - A crackling fire sound is played when the campfire (`pointLight`) is enabled.
  - The sound is spatialized (`ma_sound_set_spatialization_enabled(&fireSound, MA_TRUE)`), allowing volume and panning to change with distance from the listener.
  - Controlled similarly by `playFire()`, `stopFire()`, and volume adjustments.

AudioManager Class and Miniaudio Setup

- **Initialization:**
  - `ma_engine_init` initializes the global audio engine (`engine`).
  - Each sound (rain, thunder, fire) is loaded via `ma_sound_init_from_file(...)`.
  - If a sound fails to load, an error is printed.
- **Streaming vs. Non-streaming:**
  - The `MA_SOUND_FLAG_STREAM` flag is used for longer audio (rain, fire) to avoid loading large files fully into memory.
- **Spatialization:**
  - By default, spatialization can be on or off. For example, using `ma_sound_set_spatialization_enabled(&fireSound, MA_TRUE)`.
  - The `setXXXPosition()` methods call `ma_sound_set_position(...)` to place each sound in 3D space.
  - `updateListenerPosition()` updates the listener's position/orientation each frame via `ma_engine_listener_s` etc.
- **Volume Control:**
  - Each sound (rain, thunder, fire) has its own volume (0.0 to 1.0).
  - `increaseXXXVolume()` and `decreaseXXXVolume()` clamp the volume and optionally print the level.
  - Pressing `F2`/`F3` can raise/lower volumes for all sounds simultaneously.

Key Audio Methods

- **Rain Example** (*Play in loop*):

Listing 38: AudioManager::playRain Function

```
void AudioManager::playRain(bool loop) {
    if (!rainLoaded) {
        std::cerr << "Rain sound not loaded." << std::endl;
        return;
    }
    ma_sound_set_looping(&rainSound, loop ? MA_TRUE : MA_FALSE);
    ma_sound_start(&rainSound);
}
```

- **Listener Updates** (*Fire Spatialization*):

Listing 39: AudioManager::updateListenerPosition Function

```
void AudioManager::updateListenerPosition(const glm::vec3& position,
                                          const glm::vec3& forward,
                                          const glm::vec3& up)
{
    ma_engine_listener_set_position(&engine, 0, position.x, position.y,
        position.z);
    ma_engine_listener_set_direction(&engine, 0, forward.x, forward.y,
        forward.z);
    ma_engine_listener_set_world_up(&engine, 0, up.x, up.y, up.z);

    // Fire sound settings (attenuation model, rolloff, min distance)
```

```
    ma_sound_set_attenuation_model(&fireSound,
        ma_attenuation_model_exponential);
    ma_sound_set_rolloff(&fireSound, 1.0f);
    ma_sound_set_min_distance(&fireSound, 2.0f);
}
```

Toggling and Updating Sounds

- **Rain Toggle (Key: R):**
  - If `rainEnabled` becomes true, call `playRain(true)`.
  - Else, `stopRain()`.
- **Fire Toggle (Key: 4):**
  - Enables/disables `pointLight`.
  - If enabled, call `playFire(true)`; else `stopFire()`.
- **Thunder Integration:**
  - During a lightning flash, thunder is scheduled with a random delay.
  - `playThunder()` is called once the delay elapses.
- **Frame-by-Frame Updates:**
  1. `updateListenerPosition()` is called every frame with the camera's `position, forward, up`.
  2. Volumes can be changed via `F2`/`F3`, which call `increaseXXXVolume()` or `decreaseXXXVolume()`.

Implementation Details

- **Audio Files**: Usually stored in `sounds/` (e.g., `rain.wav`, `thunder1.mp3`, `fire.mp3`).
- **Spatial vs. Non-Spatial**:
  - *Rain* and *thunder* often use *non-spatial* (ambient-like).
  - *Fire* uses *spatial* (`ma_sound_set_spatialization_enabled(..., MA_TRUE)`) so volume/-panning change with distance.
- **User Interactions**:
  - Press `R` to toggle rain (audio + visuals).
  - Press `4` to toggle the campfire light and fire sound.
  - Use `F2`/`F3` to raise/lower volumes for all sounds.

Summary Through the `AudioManager` class and the `miniaudio` backend, the project layers multiple audio sources (rain, thunder, fire) in sync with the visual effects. Spatialization for the fire ensures positional realism, while thunder is triggered after lightning with a random delay. The resulting soundscape reinforces immersion, tying together weather, lighting, and environmental cues for a cohesive, atmospheric forest scene.

3.5.7 Fragment Discarding Effect

The fragment discarding effect adds an extra layer of realism and detail to the rendering process by selectively discarding certain fragments based on a dedicated dissolve texture. This effect is particularly useful for creating transparency, dissolving effects, or realistic alpha blending in specific materials.

- **Dissolve Texture:**
  - A separate RGBA dissolve texture is used for fragment discarding, distinct from the sRGB diffuse texture to retain automatic gamma correction.
  - The alpha channel in the dissolve texture determines whether a fragment should be discarded.
- **Texture Loading:**
  - The dissolve textures are loaded using a modified texture loading function that identifies the "Dissolve" keyword in the mtl file.
  - These textures are loaded with RGBA format to include the alpha channel needed for fragment discarding.
- **Implementation in Shader:**
  - During fragment processing, the dissolve texture is sampled at the current texture coordinates.
  - If the alpha value of the sampled dissolve texture is below a threshold (e.g., `0.1`), the fragment is discarded using the `discard` keyword:

    ```
    vec4 dissolveColor = texture(dissolveTexture, fTexCoords);
    if (dissolveColor.a < 0.1) discard;
    ```

  - This mechanism ensures that fragments with low alpha values do not contribute to the final rendered image.
- **Advantages:**
  - Separating dissolve textures from diffuse textures allows for precise control of fragment transparency while retaining gamma correction for the diffuse texture.
  - The effect is lightweight and efficient, as it avoids unnecessary processing for discarded fragments.
- **Integration with Materials:**
  - Materials that include a dissolve texture are automatically detected during model loading, ensuring seamless integration.
  - The dissolve texture path is retrieved from the material properties and applied to the respective material:

Figure 11: Fragment Discarding Effect: Transparent areas achieved by discarding fragments based on dissolve textures.

This fragment discarding effect ensures precise control over material transparency and significantly enhances the visual fidelity of the 3D scene.

3.5.8 HDR and Bloom Effects

The implementation of High Dynamic Range (HDR) and Bloom effects enhances the visual realism and dynamic lighting in the scene, creating a vivid and immersive experience.

High Dynamic Range (HDR) HDR simulates the wide range of light intensities found in real-world scenes, from very bright highlights to deep shadows.

- **HDR Framebuffer:**
  - The scene is rendered into an off-screen framebuffer (HDR framebuffer) using a high-precision floating-point format, typically `GL_RGBA16F`.
  - Two color buffers (`colorBuffers[0]` and `colorBuffers[1]`) are attached:
    1. **Main Scene Rendering Buffer** (`colorBuffers[0]`).
    2. **Bright Color Extraction Buffer** (`colorBuffers[1]`), used later for Bloom.
  - A depth attachment (`rbo`) uses a `GL_RENDERBUFFER` for depth testing.
- **Tone Mapping:**
  - After rendering into this HDR framebuffer, the resulting colors are "tone-mapped" in a post-processing pass.
  - A simple exponential mapping is used:

    ```
    // in the final pass:
    vec3 mapped = vec3(1.0) - exp(-color * exposure);
    ```

  - `exposure` controls overall brightness.
  - Gamma correction ($\gamma = 2.2$) is then applied.
- **Exposure Control:**

- An adjustable `exposure` parameter can be modified at runtime to simulate different lighting conditions (e.g., bright midday vs. dusk).
- This is passed as a uniform to the post-processing shader.

(a) Without HDR

(b) With HDR (higher exposure)

Figure 12: HDR Comparison: Enhanced lighting with High Dynamic Range rendering.

Bloom Effect Bloom enhances bright areas of the scene, creating a soft glow (as seen in real cameras) that extends around intense lights.

- **Bright Color Extraction:**
  - During scene rendering, fragments with brightness above a threshold (e.g., `1.0`) are written to `colorBuffers[1]`.

Listing 40: Brightness Calculation and Thresholding

```
float brightness = dot(finalResult, vec3(0.2126, 0.7152, 0.0722));
if (brightness > threshold) {
    gBrightColor = vec4(finalResult, 1.0);
} else {
    gBrightColor = vec4(0.0);
}
```

  - This isolates the super-bright fragments for subsequent blur passes.

- **Gaussian Blur:**
  - A two-pass blur approach is used (horizontal + vertical) repeatedly (ping-pong FBO):

Listing 41: Blur Shader Ping-Pong Loop

```
bool horizontal = true, firstIteration = true;
for (unsigned int i = 0; i < blurIterations; i++) {
    glBindFramebuffer(GL_FRAMEBUFFER, pingpongFBO[horizontal]);
    glUniform1i(horizontalLoc, horizontal);

    // bind the appropriate texture
    if (firstIteration) {
        glBindTexture(GL_TEXTURE_2D, colorBuffers[1]);
    } else {
        glBindTexture(GL_TEXTURE_2D, pingpongColorbuffers[!horizontal])
            ;
    }
    renderQuad();

    horizontal = !horizontal;
    if (firstIteration)
        firstIteration = false;
}
```

  - Repeated for `blurIterations` times (commonly 10–15) to produce a soft glow.

- **Final Composition:**
  - In the final post-processing pass, the blurred bright texture is added to the original HDR color:

```
if(bloom == 1) {
    color += bloomColor; // bloomColor from ping-pong FBO
}
```

  - Tone mapping is then applied to the combined result.

(a) Without Bloom

(b) With Bloom

Figure 13: Bloom Comparison: Soft glow effect around bright areas.

Implementation Highlights

- **Seamless Integration:**
  - HDR and Bloom are integrated into the project's deferred or forward rendering pipeline as a final post-processing step.

- **Toggle in Real-Time:**
  - Keyboard shortcuts can enable/disable HDR and Bloom. This helps test performance or visualize the difference quickly.

- **Customizable Parameters:**
  - `exposure`, `threshold`, and `blurIterations` are user-controllable for fine-tuning.
  - Adjusting `exposure` simulates the effect of changing camera aperture or overall scene brightness.

Listing 42: HDR Final Pass Fragment Shader

```
#version 410 core
out vec4 FragColor;
in vec2 TexCoords;

uniform sampler2D hdrBuffer;
uniform sampler2D bloomBlur;
uniform int bloom;
uniform float exposure;

void main()
{
```

```
    vec3 hdrColor = texture(hdrBuffer, TexCoords).rgb;
    vec3 bloomColor = texture(bloomBlur, TexCoords).rgb;
    vec3 color = hdrColor;

    if(bloom == 1) {
        color += bloomColor; // bloomColor from ping-pong FBO
    }

    // Tone mapping
    vec3 mapped = vec3(1.0) - exp(-color * exposure);

    // Gamma correction
    float gamma = 2.2;
    vec3 gammaCorrected = pow(mapped, vec3(1.0 / gamma));
    FragColor = vec4(gammaCorrected, 1.0);
}
```

Listing 43: Blur Shader (Fragment)

```
#version 410 core
out vec4 FragColor;
in vec2 TexCoords;

uniform sampler2D image;
uniform int horizontal;

void main()
{
    float weight[5] = float[](0.227027, 0.1945946,
                              0.1216216, 0.054054, 0.016216);
    vec2 tex_offset = 1.0 / vec2(textureSize(image, 0));
    vec3 result = texture(image, TexCoords).rgb * weight[0];

    for(int i = 1; i < 5; ++i)
    {
        if(horizontal == 1) {
            result += texture(image, TexCoords +
                      vec2(tex_offset.x * i, 0.0)).rgb * weight[i];
            result += texture(image, TexCoords -
                      vec2(tex_offset.x * i, 0.0)).rgb * weight[i];
        } else {
            result += texture(image, TexCoords +
                      vec2(0.0, tex_offset.y * i)).rgb * weight[i];
            result += texture(image, TexCoords -
                      vec2(0.0, tex_offset.y * i)).rgb * weight[i];
        }
    }
    FragColor = vec4(result, 1.0);
}
```

Summary By combining HDR and Bloom, the rendering pipeline captures a broader range of luminosity and accentuates bright highlights. Tone mapping ensures visually pleasing output on standard displays, while Bloom adds a glow to intense light sources. Users can dynamically toggle these effects and adjust parameters (`exposure`, blur passes, etc.) to achieve their preferred lighting style. The end result is a more cinematic and realistic forest environment, with a clear distinction between bright and dark areas and a soft halo around luminous objects.

3.5.9 Water Effect (Lake Rendering)

The water effect brings life to the scene by simulating realistic water surfaces for the lake. This implementation incorporates dynamic reflections, refractions, and wave animations to enhance the visual appeal of the lake.

Key Features

- **Reflection and Refraction:**
  - The water surface reflects the environment (skybox, nearby objects) using a reflection framebuffer.
  - A refraction framebuffer captures the distorted view of submerged objects, simulating underwater visibility.
- **Dynamic Waves:**
  - A DuDv (Distortion and Displacement) map simulates wave distortions on the water's surface.
  - A time-based `moveFactor` shifts the DuDv coordinates to animate wave motion.
- **Depth-Dependent Effects:**
  - The refraction framebuffer includes a depth texture, allowing calculation of water depth (`waterDepth`).
  - Deeper water is muddier (blending toward a `mudColor`) and less transparent.
  - Specular reflections and highlights vary with depth as well.
- **Normals and Lighting Integration:**
  - A normal map provides per-fragment normals to interact with the scene's lights (specular highlights).
  - Wave animation modifies these normals for a dynamic, shimmering water surface.
- **Fresnel Effect:**
  - Fresnel-based reflection vs. refraction weighting: at oblique viewing angles, reflection dominates; at normal incidence, refraction is stronger.
  - Implemented by dotting the view vector with the water surface normal, adjusting the blend factor.
- **Clipping Planes:**
  - During reflection rendering, a clipping plane excludes scene parts below the water plane.
  - For refraction, a second clipping plane excludes objects above the water plane.
  - This avoids artifacts and ensures only the correct portion of the scene is rendered into each framebuffer.

Figure 14: Effect of water normals on reflections and lighting.

Implementation Highlights

- **Framebuffers:**
  - `reflectionFrameBuffer` for rendering the reflection texture (`reflectionTexture`).
  - `refractionFrameBuffer` for the refraction texture (`refractionTexture`) and its depth (`refractionDepthTex`
  - Clipping planes are set via uniforms (`clipPlane`) to trim unwanted geometry above/below the water plane.
- **Shaders:**
  - **Vertex Shader**:
    - Transforms water tiles into world space.
    - Calculates texture coordinates (possibly tiled by `tiling`).
    - Computes vectors to the camera (`toCameraVector`) and from the light (`fromLightVector`).
  - **Fragment Shader**:
    - Samples `reflectionTexture` and `refractionTexture` at distorted coordinates (using the DuDv map).
    - Blends these with depth-based coloring (`mudColor`) and normal-map–based specular highlights.
    - Implements Fresnel by mixing reflection vs. refraction.
    - Outputs an alpha that can fade out at shallower areas or based on `waterDepth`.
- **Wave Animation:**
  - A `moveFactor` is incremented each frame (`moveFactor += waveSpeed * deltaTime;`).
  - Used to scroll the DuDv map and produce continuous wave movement.
  - Water depth influences wave distortion strength (`waveStrength`).
- **Real-Time Adjustments:**
  - The water tiles are rendered every frame, updated with the current camera and time-based movement.
  - Reflection/refraction passes are done first, then the main scene, then water is composited on top (in the same or subsequent pass).

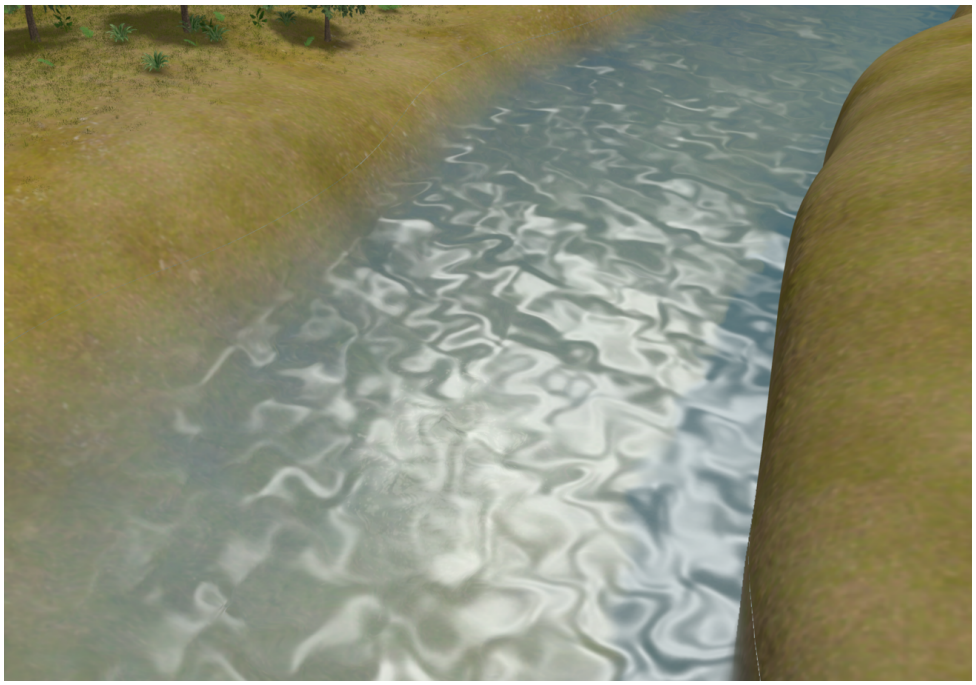


Figure 15: Specular highlights on water's surface.

Listing 44: Reflection and Refraction Framebuffers

```
    // Reflection FB: reflectionTexture + reflectionDepthBuffer (renderbuffer
        depth)
    // Refraction FB: refractionTexture + refractionDepthTexture (texture depth)
    void WaterFrameBuffers::bindReflectionFrameBuffer() {
        // Binds reflection framebuffer, viewport = REFLECTION_WIDTH x
            REFLECTION_HEIGHT
    }

    void WaterFrameBuffers::bindRefractionFrameBuffer() {
        // Binds refraction framebuffer, viewport = REFRACTION_WIDTH x
            REFRACTION_HEIGHT
    }
```

Listing 45: WaterTile Structure

```
struct WaterTile {
    float x, z, height;
    static const float TILE_SIZE;

    // ...
};
```

Holds the position/height for a water "tile" on the map.

Listing 46: WaterRenderer Class

```
class WaterRenderer {
public:
    WaterRenderer(...);
    ~WaterRenderer();
    void render(const std::vector<WaterTile>& waterTiles, ...);

private:
    gps::Shader waterShader;
    GLuint VAO, VBO, EBO;
    GLuint dudvMap, normalMap;
```

```
    float waveSpeed = 0.03f;
    float moveFactor = 0.0f;
    ...
};
```

Listing 47: Water Vertex Shader

```
#version 410 core
in vec2 position;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;
uniform vec3 cameraPos;
uniform vec3 lightPosition;

out vec4 clipSpace;
out vec2 texCoord;
out vec3 toCameraVector;
out vec3 fromLightVector;

const float tiling = 4.0;
void main()
{
    vec4 worldPos = model * vec4(position.x, 0.0, position.y, 1.0);
    clipSpace     = projection * view * worldPos;
    gl_Position   = clipSpace;

    texCoord = (vec2(position.x/2.0 + 0.5, position.y/2.0 + 0.5)) * tiling;
    toCameraVector = cameraPos - worldPos.xyz;
    fromLightVector = worldPos.xyz - lightPosition;
}
```

Listing 48: Water Fragment Shader

```
#version 410 core
in vec4 clipSpace;
in vec2 TexCoord;
in vec3 toCameraVector;
in vec3 fromLightVector;
out vec4 FragColor;

uniform sampler2D reflectionTexture;
uniform sampler2D refractionTexture;
uniform sampler2D dudvMap;
uniform sampler2D normalMap;
uniform sampler2D depthMap; // from refraction pass

uniform float moveFactor;
const float waveStrength = 0.04;
...
const vec4 mudColor = vec4(0.22, 0.35, 0.25, 0.6);

void main()
{
    // Convert clipSpace to NDC (xy / w), shift to [0,1]
    vec2 ndc = (clipSpace.xy / clipSpace.w) * 0.5 + 0.5;
    // reflection texCoord flips y
    vec2 reflectionTexCoord = vec2(ndc.x, 1.0 - ndc.y);
    vec2 refractionTexCoord = vec2(ndc.x, ndc.y);

    // Retrieve depth from refractionDepthTexture
    float floorDepth = texture(depthMap, refractionTexCoord).r;
    // ...
```

```
30
31      // Distortion via DuDv map
32      vec2 distortedTexCoords = texture(dudvMap, vec2(texCoord.x + moveFactor,
            texCoord.y)).rg * 0.1;
33      distortedTexCoords = texCoord + vec2(distortedTexCoords.x,
            distortedTexCoords.y + moveFactor);
34
35      // reflectionTexCoord += (some distortion)...
36      // clamp them
37
38      vec4 reflectionColor = texture(reflectionTexture, reflectionTexCoord);
39      vec4 refractionColor = texture(refractionTexture, refractionTexCoord);
40
41      // blend refractionColor with mudColor depending on waterDepth
42      // normal map for specular
43      // fresnel factor
44      // final out
45      FragColor = ...;
46  }
```

Rendering Procedure

1. **Reflection Pass:** Bind reflection FBO.
   - Flip camera vertically around water plane.
   - Clip plane excludes geometry below water.
   - Render scene into `reflectionTexture`.
   - Restore camera.
2. **Refraction Pass:** Bind refraction FBO.
   - Regular camera orientation, clip plane excludes geometry above water.
   - Render scene into `refractionTexture` + `refractionDepthTexture`.
3. **Main Scene Pass:**
   - Render other scene objects as usual (including skybox, terrain).
4. **Water Render:**
   - Use the final camera.
   - Bind `reflectionTexture`, `refractionTexture`, `depthTexture`, `dudvMap`, `normalMap`.
   - Draw water quads (`WaterTile`).
   - The water fragment shader fetches reflection/refraction and blends them with wave distortions, depth-based color, Fresnel, and specular highlights.

Visual Enhancements This water system adds depth and realism to the forest scene through:

- **Dynamic Reflections/Refractions**: The lake accurately reflects or refracts the environment based on viewing angle.
- **Animated Waves**: Wave motion is driven by a shifting DuDv map.
- **Lighting Integration**: With normal mapping and specular, the water surface shimmers under strong light.

Overall, these aspects produce a convincing lake surface that contributes to the immersive quality of the forest environment.

3.5.10 Optimization Techniques

To ensure optimal performance and resource utilization, several optimization techniques were implemented across different parts of the rendering pipeline. These optimizations reduce computation overhead, improve memory management, and enhance the visual quality of the scene without sacrificing performance.

Key Optimization Strategies

- **Mesh Batching**
  - *MeshBatch* objects group multiple smaller meshes into larger batches, reducing the number of draw calls.
  - Batches are organized based on shared materials and sorted to minimize state changes such as texture bindings and shader switches.
  - This significantly reduces overhead when rendering large numbers of small objects.
- **Bounding Volume Hierarchies (BVH)**
  - A BVH data structure organizes `MeshBatch` objects spatially.
  - During rendering, BVH traversal quickly identifies objects within the camera's frustum and discards those outside.
  - This hierarchical culling reduces GPU workload by limiting draw calls to only visible geometry.
- **Frustum Culling**
  - Both BVH nodes and individual object AABBs (Axis-Aligned Bounding Boxes) are tested against the camera frustum.
  - Only geometry within the frustum is rendered, skipping occluded or off-screen objects.
  - To keep culling effective, large batches are further split into sub-batches (maximum of 35,000 indices). This prevents large, broad batches from always being considered "in view."
  - This strategy is especially beneficial for dense scenes.
- **Texture Optimization**
  - Repetitive textures (e.g., tree bark, grass) are downscaled to a lower resolution.
  - This saves GPU memory and reduces texture fetch bandwidth, with minimal visual impact.
- **Vertex Deduplication**
  - Duplicate vertices are identified and merged at load time.
  - Reduces memory usage and improves cache performance during rendering.
- **Geometry Simplification**
  - The `meshoptimizer` library simplifies meshes by reducing vertex/index counts while preserving overall shape.
  - Performed on the CPU before final buffering, balancing visual fidelity and performance—particularly in complex scenes.
  - `meshoptimizer` also reorganizes index/vertex data to improve cache locality (vertex cache, overdraw, and vertex fetch).
- **Shader Optimizations**
  - Certain uniform values are cached within shader code to avoid redundant GPU calls.
  - Repeated uniform lookups are minimized, improving performance without reducing visual fidelity.

Performance Improvements

- Significantly higher frame rates from reduced draw calls and GPU load.
- Memory usage optimized through deduplication and texture downscaling.
- Better scalability—adding more objects or effects does not noticeably degrade performance.

Below is relevant code illustrating BVH construction, mesh batching, and `meshoptimizer` usage:

Listing 49: BVHNode Class (BVHNode.hpp)

```
// ============================ BVHNode.hpp
class BVHNode {
public:
    BVHNode* leftChild;
    BVHNode* rightChild;
    glm::vec3 minBounds;
    glm::vec3 maxBounds;
    std::vector<MeshBatch*> meshBatches;

    bool isLeaf() const {
        return !leftChild && !rightChild;
    }

    bool isVisible(const Frustum& frustum) const {
        return frustum.isVisible(minBounds, maxBounds);
    }
    ...
};
```

Listing 50: BVH Class (BVH.hpp)

```
// ============================ BVH.hpp
class BVH {
public:
    BVHNode* root;

    void build(std::vector<MeshBatch*>& batches) {
        if (batches.empty()) return;
        root = buildRecursive(batches, 0, batches.size());
    }

    void frustumCulledDraw(const Frustum& frustum, Shader& shader) {
        traverseAndDraw(root, frustum, shader);
    }
    ...
private:
    BVHNode* buildRecursive(std::vector<MeshBatch*>& batches, size_t start,
        size_t end) {
        if (start >= end) return nullptr;

        BVHNode* node = new BVHNode();
        // compute bounding box for all batches in [start, end)
        // if number of batches <= LEAF_SIZE, node->meshBatches = ...
        // else split half by largest axis, node->leftChild = ..., node->
            rightChild = ...
        return node;
    }

    void traverseAndDraw(BVHNode* node, const Frustum& frustum, Shader&
        shader) {
        if (!node || !node->isVisible(frustum)) return;

        if (node->isLeaf()) {
```

```
                for (auto mb : node->meshBatches) {
                    mb->Draw(shader, frustum);
                }
            } else {
                traverseAndDraw(node->leftChild, frustum, shader);
                traverseAndDraw(node->rightChild, frustum, shader);
            }
        }
        ...
    };
```

Listing 51: Mesh Optimizer Usage in Model3D.cpp

```
    // ============================ Model3D.cpp snippet (meshoptimizer usage)
    void OptimizeMesh(...) {
        // Use meshoptimizer to generate vertex remap, optimize vertex cache,
            overdraw, etc.
        // Then possibly do geometry simplification:
        size_t simplifiedIndexCount = meshopt_simplify(...);
        // Replace old vertex/index data with optimized data
    }
```

These optimizations ensure that the scene maintains immersive visual quality while running efficiently, handling large object counts and complex effects with minimal performance impact.

4 Graphical User Interface Presentation / User Manual

This section details the controls and UI elements that enable navigation, toggling of effects, and environment manipulation.

4.1 Keyboard Controls

The application supports diverse keyboard inputs for camera movement, effect toggles, and scene adjustments:

4.1.1 Camera Movement

- **W/S/A/D**: Move forward/backward/left/right.
- **Q/E**: Move down/up.

4.1.2 Rendering Modes

- **1**: Wireframe mode.
- **2**: Point mode.
- **Default**: Fill mode.

4.1.3 Visual Effects

- **B**: Toggle bloom.
- **N**: Toggle normal mapping.
- **F**: Toggle fog.
- **F4**: Toggle HDR.
- **F5/F6**: Increase/decrease HDR exposure.

4.1.4 Lighting Controls

- **3**: Toggle directional light.
- **4**: Toggle point light/campfire (+ fire sound).
- **5**: Toggle spotlight.
- **6**: Toggle vehicle headlights.

4.1.5 Environmental Controls

- **H**: Switch day/night modes.
- **R**: Toggle rain (+ rain sound).
- **7**: Toggle wind effect.

4.1.6 Waypoint and Tour Controls

- **T**: Start camera tour.
- **Y**: Stop camera tour.
- **P**: Record a waypoint.
- **O**: Save waypoints to file.
- **L**: Load waypoints from file.
- **- (Minus)**: Remove last recorded waypoint.

4.1.7 Scene Adjustment

- **Arrow Keys (Up/Down/Left/Right)**: Adjust directional light on X/Y.
- **Page Up/Page Down**: Adjust light height (Z).

4.1.8 Audio Volume

- **F2**: Increase rain, thunder, and fire volume.
- **F3**: Decrease rain, thunder, and fire volume.

4.1.9 Application Controls

- **ESC**: Exit the application.
- **F11**: Toggle fullscreen mode.

4.2 Mouse Controls

- **Mouse Movement**: Controls camera pitch (vertical) and yaw (horizontal).
- **Mouse Scroll**: Adjusts camera zoom (FOV).

4.3 Graphical Feedback

- Console logs confirm toggles (e.g., "`Bloom Toggled: ON`").
- Immediate visual changes appear (e.g., turning on spotlight or raising HDR exposure).

4.4 Dynamic Environment Interaction

- **Day/Night Transition**: Smooth color/lights blending upon toggling day/night (`H`).
- **Rain, Wind, and Lightning**: Environmental effects synchronized with audio and visual cues.
- **Waypoint System**: Record, load, or remove camera waypoints for guided tours.

4.5 Callbacks and Resizing

- Resizing the application window reconfigures HDR and water framebuffers to match new dimensions.
- Mouse input can switch to raw mode for better precision.

4.6 Summary

The interface balances ease of use and extensive control, supporting real-time toggling and parameter adjustments. Combined with audio feedback and dynamic environment changes, it provides an immersive and intuitive user experience.

5 Conclusions and Further Developments

5.1 Conclusions

This project showcases a comprehensive blend of advanced computer graphics methods in an interactive forest scene. From dynamic lighting and water rendering to real-time environmental effects (Rain, Fog, Wind) and audio synchronization, the system delivers an immersive and high-fidelity experience. By employing optimizations such as BVH-based culling and mesh batching, performance remains robust even in complex scenes. The modular design ensures scalability, providing a foundation for further experimentation with emerging graphics techniques and larger-scale environments.

5.2 Further Developments

Though the current implementation meets high standards of realism and efficiency, there are numerous avenues for future enhancements:

5.2.1 Scene Complexity

- **Expanded Terrain**: Increasing the size and diversity of the forest, possibly with procedural terrain generation.
- **Vegetation Instancing**: Adopting GPU instancing for large swaths of trees or grass, boosting performance.

5.2.2 Rendering Techniques

- **Parallax Mapping**: Introducing additional depth cues on surfaces using displacement details in shaders.
- **Deferred Shading**: Splitting geometry and lighting passes for more complex scenes with multiple lights.
- **Screen-Space Ambient Occlusion (SSAO)**: Simulating subtle contact shadows in real-time.
- **Physically Based Rendering (PBR)**: Using physically correct lighting models for materials like metal, wood, or wet surfaces.
- **Tessellation and Compute Shaders**: Dynamically refining geometry or offloading particle systems for increased detail and efficiency.

5.2.3 Optimizations and Performance Enhancements

- **Occlusion Culling**: Preventing rendering of objects hidden entirely behind others.
- **Level of Detail (LOD)**: Using meshoptimizers for distance-based geometry simplification.
- **Alternative Spatial Structures**: Experimenting with octrees or other culling strategies to handle even larger environments.

5.2.4 Animation and Interactivity

- **Skeletal Animation**: Adding animated characters or creatures.
- **Enhanced Camera Systems**: First-person or VR-based controls, plus advanced effects like head bobbing.

5.3 Closing Remarks

By exploring advanced rendering (HDR/Bloom, water reflection/refraction), dynamic environment simulations (Rain, Wind, Fire), and thorough optimization (BVH culling, mesh batching, geometry simplification), this project illustrates the depth and potential of real-time graphics. Future expansions can push this environment further toward large-scale, fully procedural worlds or adopt more physically based approaches, reflecting ongoing developments in computer graphics and interactive simulation.

6 References

- Joey de Vries, *Learn OpenGL*, `https://learnopengl.com/`. Accessed for understanding OpenGL concepts, rendering techniques, and modern graphics pipeline implementation.
- Etay Meiri, *OGLdev Tutorials*, `http://ogldev.org/`. Referenced for advanced OpenGL tutorials, including lighting, shadow mapping, and optimization techniques.
- *Laboratories of Technical University of Cluj-Napoca (UTCN)*, materials and lectures. Provided foundational knowledge and hands-on experience in computer graphics.
- *YouTube Tutorials by Professor Constantin Ioan Nandra*, focusing on Blender functionalities and techniques.