Design and Implementation of a Terrain Engine

Ivan Iazykov
Course: Fundamentals of Computer Graphics
Institution: Tsinghua University

May 25, 2025

Introduction

This project implements a terrain engine, compiled into a standalone executable. The basic goal is rendering a 3D scene that contains a skybox, water, and terrain, with an ability to fly around. Then, advanced features, such as lighting, shadows, weather and dynamic waves, add more realism into it. The engine is written in C++ and uses **OpenGL** 3.3 as its rendering backend (core profile, enabling full control over the graphics rendering pipeline), with **GLSL** for programmable shaders. The rendered scene with all effects enabled looks as follows:

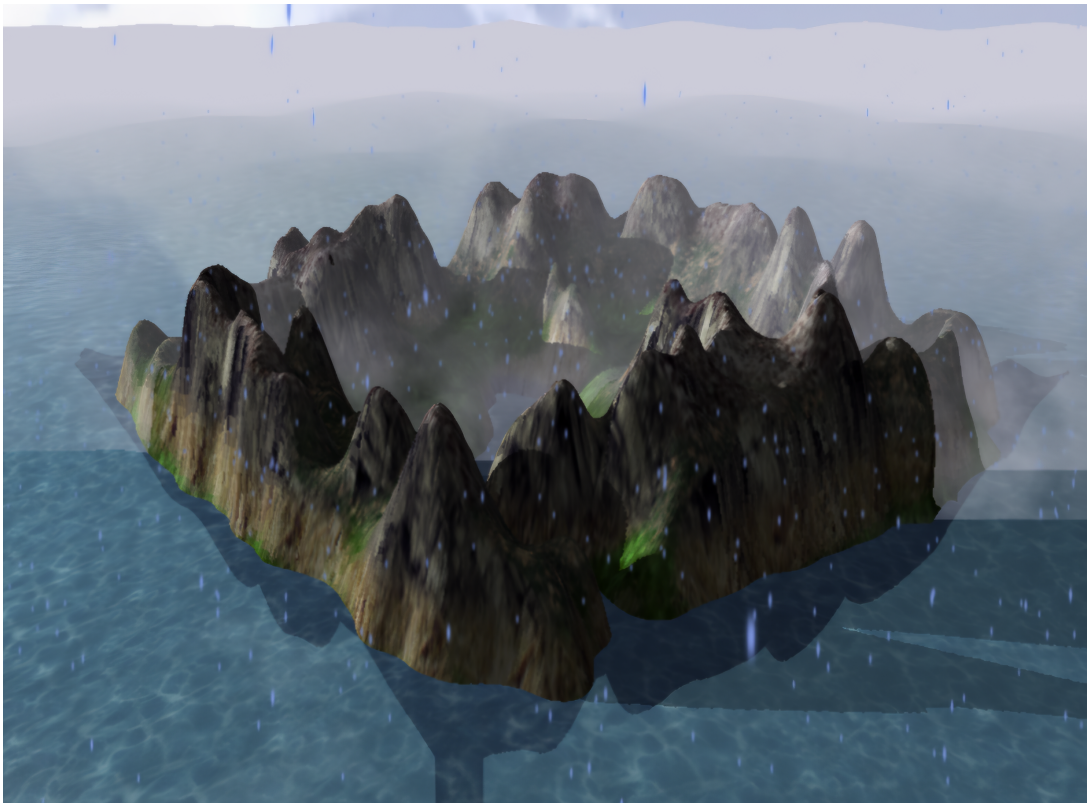

Figure 1: Rendered 3D scene

Core functionality:

- **Camera** – system that emulates first-person view camera (fly around + look around + zoom)
- **Skybox** – large cube that encompasses the entire scene and contains 6 images of a surrounding environment
- **Water** – horizontal plane with reflections and texture movement that emulates dynamic water flow
- **Terrain** – 3D mesh generated from a height map

Advanced:

- **3D waves** – Gerstner waves model
- **Lighting** – Phong lighting model (ambient + diffuse + specular lighting)
- **Shadows** – shadow mapping
- **Weather** – particle emitters for fog and rain, distance based fog effect
- **Smooth camera movement** – acceleration / deceleration
- **Fullscreen mode support**

1 Skybox

The Skybox entity is implemented in `skybox.h`. This is our first step in building the scene. A *skybox* is a large cube that encompasses the entire scene and displays six images of the surrounding environment. It is centered on the player (always at the skybox's origin (0, 0, 0)) and moves with him, giving the player an illusion that the environment he's in is much larger than it actually is. Our skybox will feature a blue sky, clouds, and the sun. However, there is no image for the bottom of the cube, because we are planning to add a water surface there later.

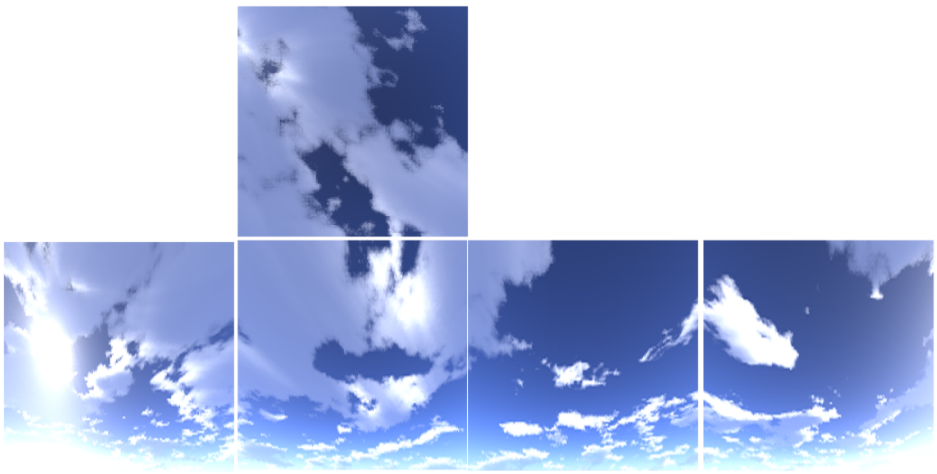

Figure 2: Images used for skybox

In OpenGL, a skybox is implemented as a cubemap texture. Its most powerful property is that you can sample the texture coordinates using only a direction vector — its magnitude doesn't matter (local vertex position on the surface of the cube = direction vector = texture coordinate). Because skybox is a 3D object, it has its own VAO, VBO, and shaders. However, rendering it correctly requires a few tricks. First, set the depth function to `GL_LEQUAL` and, in the skybox's vertex shader, force $z = w$ so that after perspective division the depth value is always 1.0 (the far plane). This ensures the skybox always appears in the background: it will fail the depth test wherever another object is closer (its depth is set to 1.0 in the vertex shader, so we need less or equal depth function). Second, remove the translation component from the view matrix, creating an illusion of an infinitely distant environments.

Hyperparameters:

`skyboxScaleRatio` – used to correct skybox, so that the sun is a circle

2 Water

The Water entity is implemented in `water.h`. Once the sky is in place, it's time to add a water surface. Implementing it as a static, flat plane is straightforward: because the water texture has low resolution, we create a grid and repeat the texture many times by setting `GL_REPEAT` texture wrapping property. To simulate movement, we add a dynamic offset to the texture coordinates each frame. To enhance realism, we then add reflections of the sky and terrain (described next). Skybox reflections are easy: we use the water surface normal and camera direction to compute a reflection vector, which is used as a direction vector for sampling the skybox. Terrain reflections are more challenging because the terrain lacks the cubemap's sampling convenience and has complex geometry. In OpenGL, the way to do it is to render the terrain from a different viewpoint — mirrored camera view. The render loop would do 2 passes in each iteration: first pass for rendering the scene from the reflection camera's point of view (mirrored over the water plane), capturing reflection image into texture using framebuffer; second pass for rendering the scene from the main camera's point of view, using this texture from the first pass. This method creates planar terrain reflection.

Hyperparameters:

`waterLevel` – world-space y-axis position of the water surface
`waterHorizontalScale` – scaling factor for the x and z axis
`GRID` – water surface grid size
`worldStep` – distance between neighboring grid point in a world space (dx = dz)
`waterSpeed` – water texture movement speed
`waveAmp`, `waveFreq`, `waveSpeed` – Gerstner wave settings
`terrainReflectionStrength`, `skyboxReflectionStrength` – reflection settings

3 Terrain

The Terrain entity is implemented in `terrain.h`. It is an island that we position somewhere on the water's surface. Three images are used to construct the terrain: *heightmap* – a grayscale image in which pixel intensity encodes height; color texture – gives terrain the mountain, grass, etc.; detail texture – adds fine-grained surface detail. First, we load the heightmap. We can already generate a complete 3D mesh from it (which can be displayed, for example, as points; see Fig. 3). This mesh is structured as triangles, which is optimal for rendering a terrain surface: each quad $\rightarrow$ 2 triangles $\rightarrow$ 6 vertices. By iterating over each pixel of the heightmap, we compute the vertex positions using the formula:

$$\text{vertex}[i, j] = (x, y, z) = (i,\ \text{heightmap}[i, j],\ j)$$

Next, we load the color and detail textures. This is straightforward: we simply pass them to the terrain's fragment shader, where the detail texture is multiplied by a detail-level factor to tile it repeatedly. The two textures are then summed together, with a constant term to adjust overall brightness.

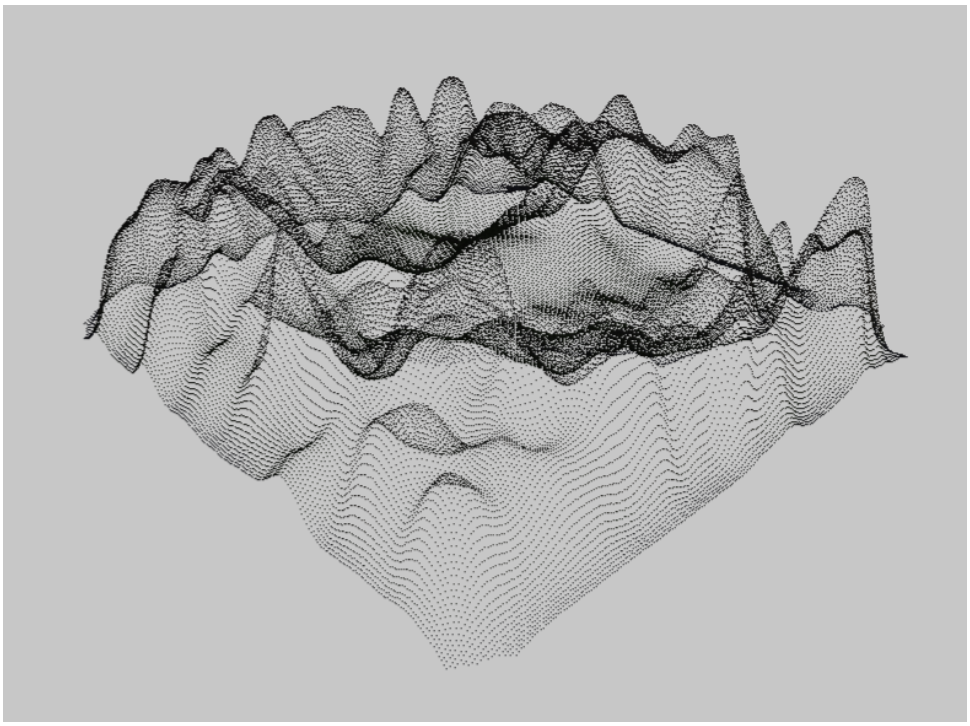

Figure 3: Terrain mesh

Hyperparameters:

`terrainOffset` – world-space y-axis position of the terrain
`terrainHorizontalScale` – scaling factor for the x and z axes
`terrainVerticalScale` – scaling factor for the y-axis + pixel value conversion
`detailLevel` – frequency of the detail texture, controlling how much detail is applied to the surface

4 Advanced

4.1 3D waves

To add more realism, we can create a fully dynamic 3D water mesh that is recalculated each frame. I chose *Gerstner waves* — a model for realistic ocean waves commonly used in games. Mathematically, they are exact solutions to the Euler equations for periodic surface gravity waves. I'm using an advanced variant that simulates wave motion in two directions (x and z), producing a two-directional Gerstner wave:

$$\mathbf{v}(x,z,t) = \begin{pmatrix} x + A\,\cos\big(k\,x - \omega\,t\big) \\ A\,\sin\big(k\,x - \omega\,t\big) \;+\; A\,\sin\big(k\,z - \omega\,t\big) \\ z + A\,\cos\big(k\,z - \omega\,t\big) \end{pmatrix}$$

, where

$\mathbf{v}(x,z,t)$ — displaced vertex position at (x,z) and time t

A — wave amplitude (controls both vertical and horizontal displacement)

k — wave number (controls spatial frequency, $k = 2\pi/\lambda$)

ω — angular frequency (controls wave speed)

One thing to point out is that, to enhance realism, I'm also computing wave normals to produce natural-looking water reflections. To compute the normal vector of a vertex, we take his main neighbors, and then use central difference derivatives $\frac{\partial y}{\partial x}$ and $\frac{\partial y}{\partial z}$:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

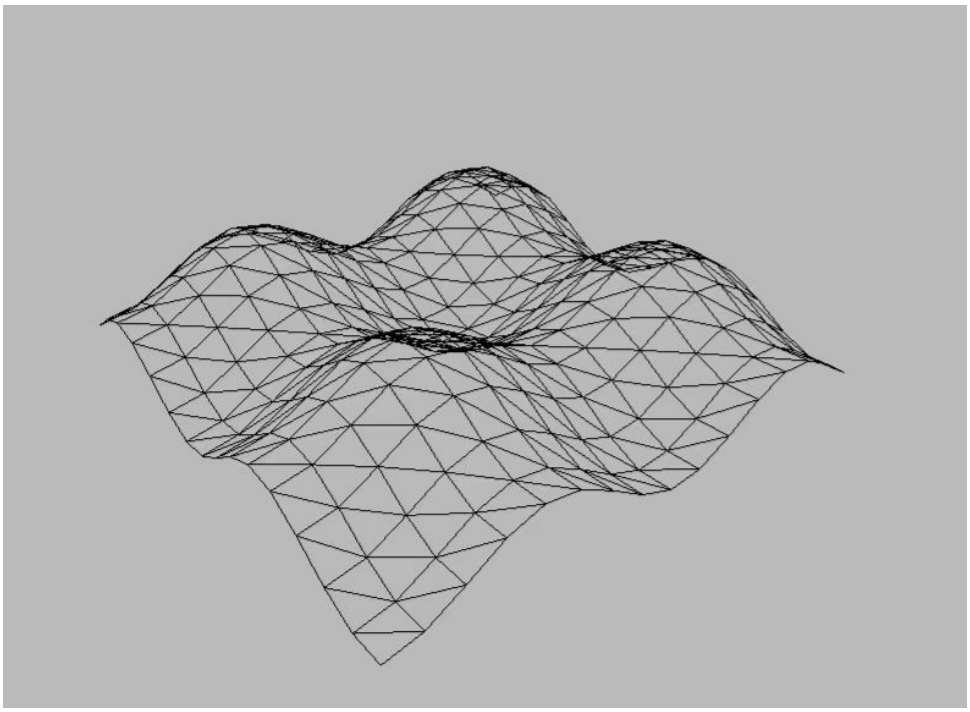

Figure 4: Gerstner wave

4.2 Lighting

Another way to enhance realism is to simulate lighting effects. They can take different form; three main ones are combined into the Phong lighting model : ambient light, diffuse light, and specular light. I introduced these lighting concepts in my Homework 6 report. In our scene, the light is positioned to represent the sun (visualized as a light cube) and applied to both the water and the terrain. Each entity has its own illumination settings, such as ambient and diffuse strength factors. All light-effect calculations are performed in the fragment shader in world space coordinates.

$$\text{ambient} = k_a \cdot \text{lightColor}$$
$$\text{diffuse} = k_d \cdot \text{lightColor} \cdot \max\left((\text{N}, \text{L}), 0\right)$$
$$\text{specular} = k_s \cdot \text{lightColor} \cdot \left(\max\left((\text{I}, \text{R}), 0\right)\right)^{\alpha}$$
$$\text{resultColor} = (\text{ambient} + \text{diffuse} + \text{specular}) \cdot \text{baseColor}$$

, where

k_a, k_d, k_s — ambient, diffuse, and specular reflection coefficients, respectively
N — entity surface normal vector
I — camera view direction vector
L — light direction vector
R — light reflection vector
α — shininess exponent (controls the size and sharpness of the specular highlight)

4.3 Shadows

I am only rendering shadows of the terrain, cast onto the terrain itself and the water. Shadows complement the lighting model. *Shadow mapping* is a technique that first renders the scene from the light's point of view to record depth values into a texture (the "shadow map"), then uses that map in the main pass to test whether each fragment is occluded from the light. Their setup and mechanics are similar to reflections in that both require an extra render pass and use a framebuffer-backed texture. But compared to reflections, shadows are not view-dependent (they only care about the light's view) and they affect the entire scene, so we use a dedicated, depth-only shader for that pass. The resulting depth texture is then bound in the entity's fragment shader, where we compare the fragment's light-space depth against the depth stored in the shadow map: if the fragment is farther from the light than what's in the shadow map, it's in shadow. Then, we modulate diffuse light component: diffuse lighting is only applied if the fragment is not in shadow (this allows for rendering shadowed areas). This is a very efficient and general approach to casting hard shadows from arbitrary geometry.

4.4 Weather

Weather is a great addition for making the terrain engine more dynamic. I implemented rain and fog in `weather rain.h` and `weather fog.h`, respectively, as particle emitters, and chose billboarding for an easy yet realistic effect. In both systems, I spawn particles around the camera with a bias toward the center: the spawn angle is random in $[0, 2\pi]$ and the distance from the camera is random in [0, `weatherRadius`]. The particle

concept was introduced in my Homework 3 report. For rain, raindrops spawn at a predefined height plus a small random offset, and a particle dies when it reaches the water level. To enhance realism, drops closer to the camera are rendered larger. For fog, particles spawn at the water level, drift horizontally in a random direction, and expire when their lifespan reaches zero. I added distance-based fog in the water's fragment shader: it blends the fog color with the water color, using a mix factor that increases with distance from the camera.

4.5 Smooth camera movement

This requires fundamental changes to the standard camera system interface, implemented in `camera.h`. We need to define two vectors: a temporary input direction — representing the camera movement direction in world space based on the current keyboard input; and a movement direction, which accumulates the camera's actual movement in world space. When input is received, only the input direction vector is updated. The actual movement logic is handled separately, allowing the camera to continue moving even when no keys are pressed: the movement direction is updated every frame, regardless of keyboard input. The camera accelerates to a maximum speed when the input direction is non-zero, and decelerates back to zero when there is no input. One important detail is that normalization must be applied to the input direction to ensure it has unit length. This keeps the movement speed consistent, regardless of how many direction inputs (keys pressed) summed into input direction.

4.6 Fullscreen mode support

This is a simple but essential feature. To enable fullscreen mode, we first retrieve the primary monitor and its video mode (info like resolution, color depth, refresh rate). Then, using `glfwSetWindowMonitor`, fullcreenmode will be on. To return to windowed mode, the function is called again, restoring window to default position + size. A problem might occur when rendering reflections and shadows: they each use a fixed resolution, so during the reflection and shadow passes the viewport must match those resolutions. This is implemented by temporarily rescaling the scene for rendering reflections / shadows, and coming back to the current window resolution. A variable for storing the current window size has to be used though.

5 Project Manual

The project can be cloned from my GitHub repository:
`git clone git@github.com:fata1error404/tsinghua-cg-spring-2025.git`

Install dependencies
`sudo apt-get install mesa-utils mesa-common-dev libglu1-mesa-dev freeglut3-dev` – core OpenGL utilities (libraries for rendering, handling windowing and output)

`sudo apt-get install libglew-dev libglfw3-dev libglm-dev` – advanced development libraries (GLEW, GLFW, GLM)

Compile and launch

```
g++ main.cpp -o app -lglfw -lglad
./app
```

Keyboard controls:

W A S D – camera movement

- + – wave height control

N – enable / disable weather

L – enable / disable lighting

M – show / hide light cube

F – fullscreen mode

Escape – exit