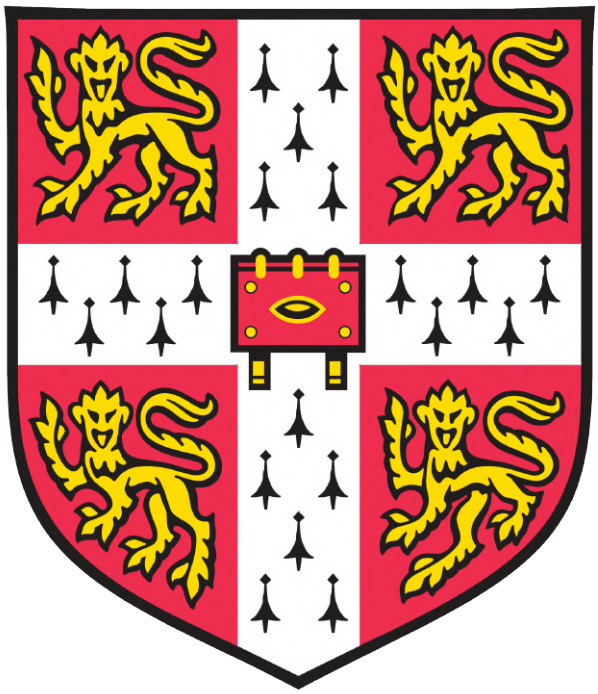
Qiaozhi Lei

Procedural Generation of Complex Terrain with Implicit Representation and Ray Marching



Computer Science Tripos – Part II
Fitzwilliam College

May 13, 2024

Declaration

I, Qiaozhi Lei of Fitzwilliam College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose. In preparation of this dissertation I did not use text from AI-assisted platforms generating natural language answers to user queries, including but not limited to ChatGPT. I am content for my dissertation to be made available to the students and staff of the University.

Signed: Qiaozhi Lei
Date: May 13, 2024

Proforma

Candidate number:	**2395B**
Project Title:	**Procedural Generation of Complex Terrain with Implicit Representation and Ray Marching**
Examination:	**Computer Science Tripos – Part II, 2024**
Word Count:	**11274** [1]
Code Line Count:	**6498** [2]
Project Originator:	**The Dissertation Author**
Project Supervisor:	**Joseph March**

Original Aims of the Project

The original aims of the project was to design and build an application capable of rendering procedurally generated natural environments in real-time using ray marching of implicit representations. The environment should include terrain, foliage, water, sky, and clouds. Users will be able to customize the environment's properties and rendering parameters, navigate the scene, and simultaneously visualize the results in a viewport.

Work Completed

This project was successful in meeting its objectives. I designed and built an application capable of generating and rendering scenes of natural environments, including terrain, trees, water, a physically-simulated atmosphere, and volumetric clouds. Additionally, I extended these techniques to create and render procedural planets. The renderer uses ray marching to find intersections with implicit scene representations accurately and efficiently. The application enables users to specify, save, and load parameters, and navigate the scenes in real time. The naturalness of the terrain and atmosphere, along with the correctness and performance of the application, were thoroughly evaluated and discussed in detail.

Special Difficulties

None.

[1] This word count was computed using `texcount`.

[2] This code line count was computed using `cloc-2.00.exe`.

Notational Conventions

Throughout this dissertation, I use the following notational conventions:

1. **Points in Space** Bold uppercase, e.g., $\mathbf{A}$, $\mathbf{B}$.
2. **Vectors**
   (a) Bold uppercase for difference vectors, e.g., $\mathbf{AB}$.
   (b) Bold lowercase for directional vectors, e.g., $\mathbf{x}$ for a normalized direction.
3. **Scalars** Lowercase or uppercase italicized, e.g., a, A, b, B for magnitude values.
4. **Vector Swizzling** Dot notation for components, e.g., if $\mathbf{P} = (\mathbf{P}_x, \mathbf{P}_y, \mathbf{P}_z)$, $\mathbf{P}.x$ for $\mathbf{P}_x$, $\mathbf{P}.yx$ for $(\mathbf{P}_y, \mathbf{P}_x)$

Key values:

1. $\mathbf{C}$: Camera position.
2. $\mathbf{r_c}$: Camera ray direction.
3. $\mathbf{r_s}$: Sun ray direction.
4. $\mathbf{c}$: Color.

Coordinate system:

Throughout this work, a coordinate system where the y-axis points upwards is used for all spatial references and calculations.

Contents

1 Introduction

1.1 Motivation

Procedural generation and rendering of natural environments has long been a valuable but challenging problem in computer graphics. It's especially useful in the entertainment industry, as manually sculpting terrain, placing trees, and creating skyboxes is labor-intensive. This task becomes even more challenging for real-time applications like video games, given the scale and level of detail in natural environments.

This project aims to design and build an application capable of rendering procedurally generated natural environments in real-time using ray marching of implicit representations. The environment should include terrain, foliage, water, sky, and clouds. Users will be able to customize the environment's properties and rendering parameters, navigate the scene, and simultaneously visualize the results in a viewport.

1.2 Project Aims

This project has the following goals:

1. Explore procedural generation techniques for creating natural terrains using implicit representations.
2. Investigate ray marching techniques to accurately and efficiently find intersections.
3. Expand terrains with details such as foliage, water, clouds, and atmosphere to produce immersive natural environments.

By integrating these goals, my objective is to develop a final product: an open-source application that serves as an interactive learning tool or starting point for students, researchers, and artists interested in procedural generation, implicit representations, and ray marching. The desired state of the application window and its components is illustrated in Figure 1.1, while sample renders created by this final state are shown in Figure 1.2 and 1.3.

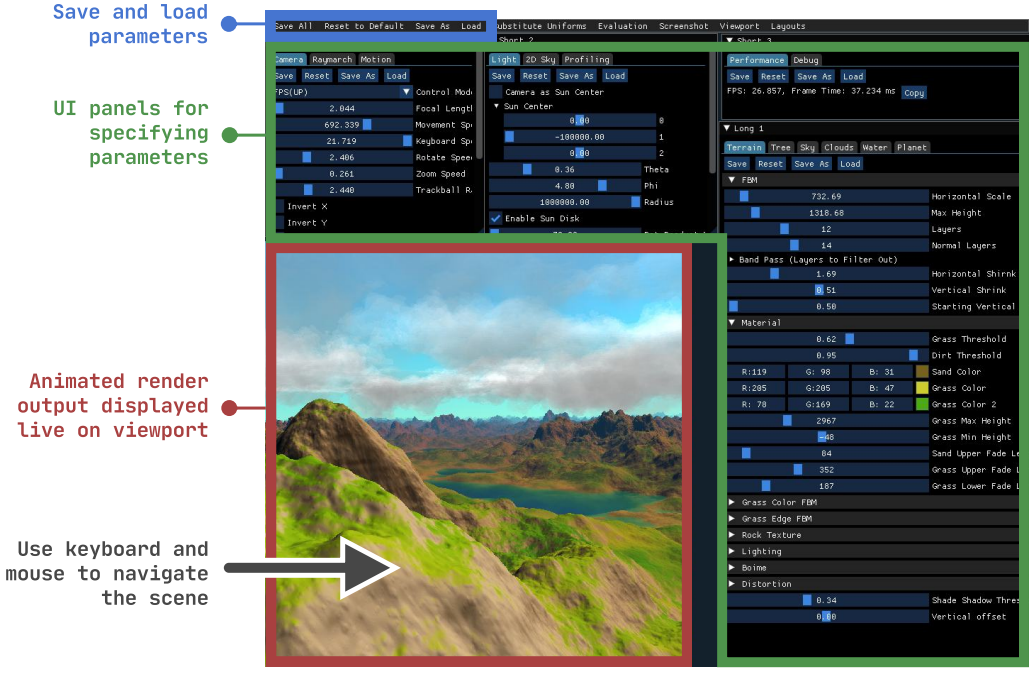


Figure 1.1: Goal state for the application window.

Figure 1.3: Example renders of natural environments in my application.

1.3 Previous Work

Terrain generation methods are typically categorized into procedural techniques, physical simulations, and example-based methods [1]. **Procedural techniques** use fractal models like fractional Brownian motion (fBm) [2] to emulate the self-similarity of terrain by combining base functions such as midpoint displacement, diamond-square algorithms [3], and Perlin noise [4], with varying frequencies and amplitudes. **Physical simulation** approaches focus on modeling processes like hydraulic erosion [5] and thermal erosion [6]. **Example-based methods** utilize real-world heightmaps and have recently incorporated Generative Adversarial Networks (GANs) for realistic textured terrain generation [7].

This project explores fBm-based procedural techniques for generating terrain heightmaps. For rendering heightmaps, common methods include converting them to meshes via algorithms like Marching Cubes [8]. Alternatively, this project employs GPU-based ray marching techniques [9] to render the heightmaps directly.

Besides terrain, other essential elements in natural environment scenes include foliage, water, clouds, and the sky. For **foliage**, while billboard techniques [10] offer a simplified representation, they lack volume; more detailed but computationally intensive methods include shape grammars and rule-based techniques, such as L-systems [11]. This project opts for an intermediate approach, employing noise-distorted Signed Distance Fields (SDFs) [12] of ellipsoids for foliage representation. For **water** simulation, while some games use animated grid meshes to mimic waves [13], this project uses a simpler method of modeling the water surface as a plane enhanced with textures and normal mapping to simulate waves [14, 15]. **Clouds and sky** are often effectively represented using skyboxes [16], although this lacks interactivity. Alternatively, clouds can be generated using fBm and rendered with volumetric techniques [17] and sky color can be derived from the physical simulation of Rayleigh scattering of atmospheric particles [18]. These are the methods employed in this project.

2 Preparation

Following the workflow of the application shown in Figure 2.1, this chapter covers background materials related to *procedural generation techniques* (Section 2.1) for creating implicit representations of scenes, and *rendering methods* (Section 2.2) for these implicit representations. Additionally, the chapter describes shaders and Graphics Processing Unit (GPU) architecture (Section 2.3), crucial elements since procedural generation and rendering are performed in fragment shaders. The chapter then describes the requirements (Section 2.4) and software engineering techniques (Section 2.5).

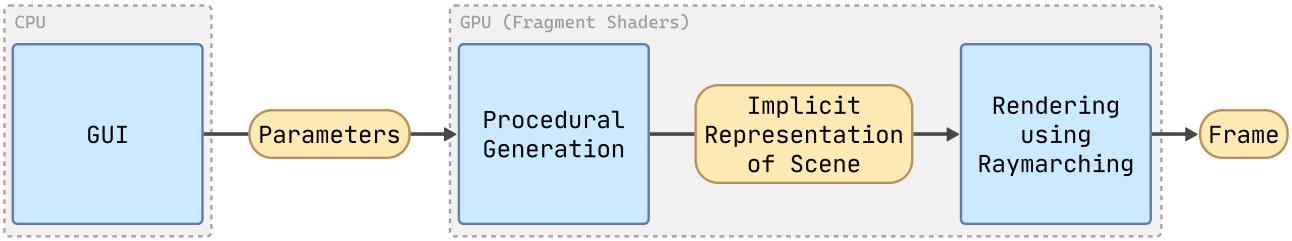


Figure 2.1: The application's workflow. Parameters are adjusted in the GUI and passed to fragment shaders as uniforms. Next, these parameters drive the procedural generation stage, creating the scene's implicit representation, which is rendered through ray marching.

2.1 Procedural Generation of Natural Environments

This section provides background materials necessary for addressing the topic of procedurally generating and representing natural elements digitally. It starts by examining nature's properties and how fractional Brownian motion (fBm) captures them, then discusses representing complex surfaces like terrains and clouds with implicit representations.

2.1.1 Properties of Nature

Understanding nature is the first step in simulating and procedurally generating it. Nature, inherently random yet ordered, offers a complex study canvas. In forests, tree placement does not follow rigid patterns, and no two leaves are identical. Yet nature maintains structure: clouds, though chaotic, follow familiar shapes rather than resembling pure noise. This blend of randomness and pattern is central to nature, often exemplified by self-similarity. Clouds, mountains, and tree bark all show this feature, where parts mimic the whole in shape and texture. Additionally, natural transitions, like the gradual shift in soil salinity from the beach to inland, reveal the continuity and smoothness of change. To effectively procedurally generate nature, it's crucial to replicate this interplay of **randomness**, **self-similarity**, and **continuity**.

2.1.2 fBm Captures the Properties of Nature

To model the properties of natural elements, I chose **fBm** in this project. fBm extends classical Brownian motion, introducing long-range dependence through the Hurst exponent [2]. fBm is characterized by self-similarity [19], meaning its statistical properties are consistent across different scales. Similar to classical Brownian Motion, fBm's paths are continuous and integrate randomness through Gaussian distribution [2]. These properties make fBm suitable for procedural generation of natural elements such as terrain and clouds.

To construct fBm, I employed fractal noises [20], specifically combining multiple layers of smooth gradient noise such as value noise (Appendix B.1). First, define $n \in \mathbb{Z}^+$ layers of

value noise, each with a specific frequency (f_i), amplitude (a_i), rotation ($\mathbf{R_i}$), and translation ($\mathbf{o_i}$), defined as:

$$f_i = \alpha^i \cdot f_0,\ a_i = \beta^i \cdot a_0,\ \mathbf{R_i} = \mathbf{R}^i,\ \mathbf{o_i} = i \cdot \mathbf{o} \tag{2.1}$$

Finally, do a point-wise summation of these layers. Let $N_k(\mathbf{P})$ represent the value of the k-dimensional noise function at point $\mathbf{P}$. A n-layer fBm at point $\mathbf{P}$ is then given by:

$$\text{fBm}_k(\mathbf{P}) = \sum_{i=0}^{n} a_i \cdot N_k(f_i \cdot \mathbf{R}_i\mathbf{P} + \mathbf{o}_i) \tag{2.2}$$

This result embodies the essential characteristics of fBm—the smooth gradient noise provides randomness and continuity, and layering noises with different frequencies creates self similarity.

2.1.3 Implicit Representations

How can large or infinite complex surfaces such as terrains and clouds be represented? This section focuses on implicit representation as a key solution, and describes two special types of implicit representations: heightmaps and Signed Distance Fields (SDFs).

Implicit Representation: A Solution for Complex Surfaces

In digital representation of 3D geometry, explicit representations like polygon meshes have traditionally been preferred, relying on vertices, edges, and faces to define surfaces. While well-supported in hardware and software, these methods face limitations with large or complex surfaces, leading to high storage and memory demands [21]. Crucially, they also lack the ability to achieve infinite resolution [21], which is vital for rendering detailed and expansive landscapes.

In contrast, implicit surface is a powerful alternative. At its core, an implicit equation is a relation in the form of $R(x_1, \ldots, x_n) = 0$, where R is the implicit function [22]. The solution space of a n-dimensional implicit equation describes the surface of an n-dimensional shape. This allows complex surfaces such as large or even infinite terrains to be defined using a single function. This approach significantly reduces memory and storage requirements compared to traditional mesh-based methods. Additionally, the sign of the implicit function indicates whether the point is inside or outside the surface (negative when inside) [22], which is convenient for ray marching.

Heightmaps

A heightmap $H(x, z)$ defines the elevation y of the terrain at each point (x, z). The implicit equation representing terrain's surface is expressed as:

$$y - H(x, z) = 0 \tag{2.3}$$

Signed Distance Functions

Another implicit representation is the SDF [12]. For each point in space, the SDF assigns a value indicating the shortest distance to the surface. The sign of this value determines if the point lies inside or outside the geometry. The implicit equation for an SDF-based surface is:

$$S(x, y, z) = 0 \tag{2.4}$$

2.2 Rendering

Ray marching is a per-pixel rendering technique. For every pixel on the screen, a **camera ray**, $\mathbf{R}(t) = \mathbf{C} + t \cdot \mathbf{r_c}$, is emitted from the camera and directed towards the respective pixel. Here,

$\mathbf{C}$ is the world coordinates of the camera, $\mathbf{r_c}$ is the normalized direction vector pointing from the camera towards the pixel, and t is the distance along the ray.

For every pixel, the rendering process is:

1. find the nearest intersection between the camera ray and scene objects.
2. apply textures and lighting based on information at the point of intersection to produce the color of the pixel.

Following this process, this section first introduces ray marching, which is employed to find intersections, and then describes some illumination and shadow models used in this project.

2.2.1 Ray Marching SDFs

One of the most efficient techniques for rendering scenes composed of SDFs is an algorithm known as **Sphere Tracing** [23]. In this context, consider a camera ray $\mathbf{R}(t)$, and the scene is assumed to contain multiple SDFs, denoted as $\text{SDF}_1, \ldots, \text{SDF}_n$.

The core idea of Sphere Tracing is to iteratively move along the ray, incrementing the distance t by the smallest distance to any surface in the scene, as illustrated in Figure 2.2. The process is expressed as:

$$t_{i+1} = t_i + \min_i(\text{SDF}_i(\mathbf{R}(t_i))) \tag{2.5}$$

This process continues until the ray either exceeds a predefined maximum distance, t_{max}, or when a specified maximum number of steps is reached, suggesting no intersection within the scene's bounds. The process also terminates when the minimum SDF at the current ray position is less than a small threshold ϵ, implying an intersection.

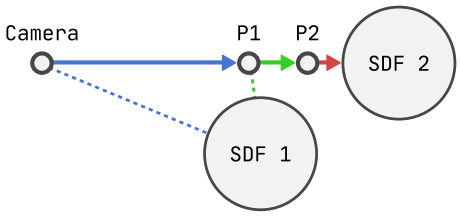


Figure 2.2: Illustration of Sphere Tracing for Ray Marching SDFs. The ray progresses until it closely approaches the boundary of SDF 2, indicating an intersection.

2.2.2 Ray Marching Heightmaps

When dealing with a heightmap H, the Sphere Tracing algorithm is inapplicable since heightmaps are not SDFs. Here, a more rudimentary form of ray marching, termed the **fixed-step ray marching algorithm** [9], is employed.

This algorithm advances the ray with a constant step size, λ:

$$t_{i+1} = t_i + \lambda \tag{2.6}$$

At each step, the algorithm evaluates the implicit function $y - H(x, z)$. It terminates when this function's value drops below zero, indicating an intersection with the heightmap. To refine the exact point of intersection, linear interpolation is used between the last two points, expressed as:

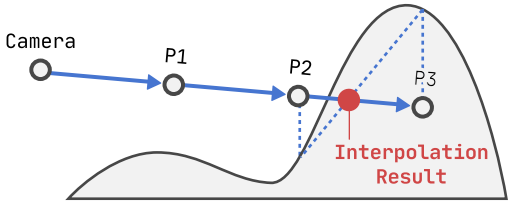


Figure 2.3: Illustration of Fixed-step Ray Marching for Heightmaps.

$$t' = t - \lambda \frac{H(\mathbf{R}(t).xz) - \mathbf{R}(t).y}{H(\mathbf{R}(t).xz) - \mathbf{R}(t).y - H(\mathbf{R}(t-\lambda).xz) - \mathbf{R}(t-\lambda).y} \tag{2.7}$$

Here, t is the distance when the camera ray first went below the heightmap, and t' is the

distance after interpolation.

This dissertation further explores several optimizations of this algorithm in Section 3.2.2. Figure 2.3 illustrates the fixed-step ray marching algorithm in action, where it finds the intersection between the camera ray and a 1D heightmap.

2.2.3 Illumination and Shadows

This section outlines the process for computing the color of a pixel once an intersection with a scene object is identified. The process applies textures and calculates lighting to determine the final color for each pixel. In this project, the material colors are determined procedurally, which is detailed in 3.2.3. Given the project's scope and focus, a simplified lighting model suffices. Therefore, I used the Phong reflection model [24], together with Fresnel reflectance and soft shadows.

Fresnel Reflectance

The Fresnel Reflection principle describes how the amount of light reflected by a surface varies with the angle of incidence [25]. The Fresnel equations provide a way to calculate the reflection coefficient, which determines the ratio of reflected light at different angles of incidence. Using the Schlick's approximation [26], the Fresnel reflectance at normal incidence, F_0, and at incident angle θ, $F(\theta)$, are given by:

$$F_0 = \left(\frac{n_1 - n_2}{n_1 + n_2}\right)^2 \tag{2.8}$$

and

$$F(\theta) = F_0 + (1 - F_0)(1 - \cos(\theta))^5 \tag{2.9}$$

Here, n_1 and n_2 are the refractive indices of the medium from which the light is coming and the medium it is entering, respectively.

Shadows

Hard Shadows The simplest shadow calculation is the binary determination of whether light reaches a point or not, resulting in hard shadows with distinct, sharp edges. In ray marching, a shadow ray is cast from the point of intersection towards the light source, and if any object is encountered, the point is deemed to be in shadow.

Soft Shadows In the physical world, shadow edges are often soft due to the finite size of light sources like the Sun, resulting in shadows with an **umbra**—the fully shadowed region—and a **penumbra**—the partially illuminated area.

A technique to simulate this involves adjusting the shadow contribution based on the nearest distance to the geometry encountered by a shadow ray, d_{min}. Ray marching can approximate this distance, as demonstrated in Figure 2.4, with a more detailed explanation in Section 3.2.4 and 3.3.4.

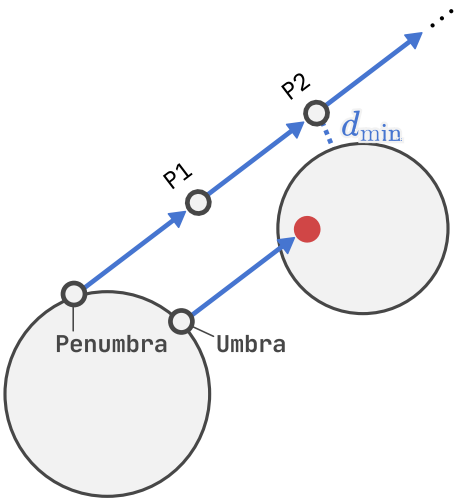


Figure 2.4: Illustration of soft shadow calculations. The shadow ray at the penumbra point does not intersect any objects, with d_{min} occurring at $\mathbf{P}_2$.

2.3 Shader Execution

Unlike traditional programming executed on the Central Processing Unit (CPU), shader programs are executed on the Graphics Processing Unit (GPU). This execution model leverages the GPU's architecture, which is designed for parallel processing, featuring hundreds or thousands of small processing cores. These cores are grouped into larger units, often called streaming multiprocessors (SMs) or compute units (CUs), as illustrated in Figure 2.5.

In graphics pipelines, there are several types of shaders:

- **Vertex Shaders**: Process vertex data and determine vertex positions in screen space.
- **Fragment Shaders**: Calculate the color and other attributes of each pixel. In this project, the procedural generation and rendering processes are implemented in fragment shaders.
- Geometry shaders and compute shaders are not used in this project.

Workload Distribution When a scene is being rendered, it is divided into fragments. The GPU's control unit assigns groups of these fragments (often in a grid pattern, such as 2x2 "pixel-quads") to the different processing cores. This distribution ensures that the workload is balanced across the GPU's cores, as illustrated in Figure 2.5.

Efficient Parallelisation Many GPU architectures use a SIMD (Single Instruction, Multiple Data) approach. In this model, each core executes the same instruction at the same time but on different data (fragments). This is efficient for fragment shaders because each fragment often undergoes similar processing.

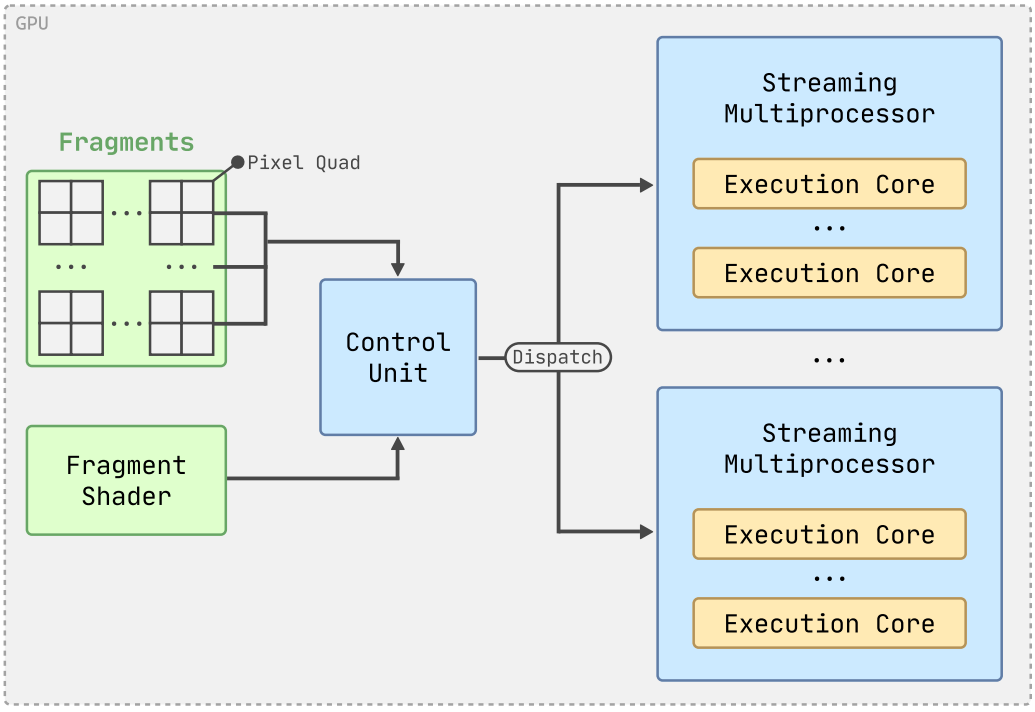


Figure 2.5: Schematic of GPU parallel processing architecture. Fragments are organized into pixel quads, and the Control Unit dispatches tasks for executing fragment shaders to multiple Streaming Multiprocessors and their respective cores.

2.4 Requirement Analysis

2.4.1 Success Criteria

The project's success criteria (Appendix H) forms the requirements:

- Achieve real-time ray marched rendering of procedurally generated natural terrains, encompassing shadows, shading, and coloring.
- Attain realistic detailing in the terrains, complemented by an authentic backdrop of clouds and sky.
- Develop a desktop application that offers camera controls, allows real-time parameter adjustments, and supports saving and loading of parameters.

2.4.2 MoSCow Analysis

I then conducted **MoSCow analysis** [27] (Must have, Should have, Could have, Won't have this time) to refine the requirements into deliverables and give priorities to the deliverables, as detailed in Table 2.1. Core deliverables are in the Must-Have category, while extensions are in the Should-Have or Could-Have categories.

Type		Deliverable	MoSCoW Analysis	Priority	Completed
Procedural Generation and Rendering of	Terrain	Heightmap Generation	Must-Have	Core	✓
		Ray Marching	Must-Have	Core	✓
		Procedural Coloring	Must-Have	Core	✓
		Illumination and Soft Shadows	Must-Have	Core	✓
	Tree	SDF Generation	Must-Have	Core	✓
		Ray Marching	Must-Have	Core	✓
		Procedural Coloring	Must-Have	Core	✓
		Illumination and Soft Shadows	Must-Have	Core	✓
	Water	Procedural Coloring and Transparency	Should Have	Extension	✓
		Reflections	Could Have	Extension	
	Atmosphere	Basic Implementation (Gradient Sky)	Must-Have	Core	✓
		Simulation of Rayleigh Scattering	Should Have	Extension	✓
	Clouds	Basic Implementation (2D Clouds)	Must-Have	Core	✓
		Density Map Generation	Should Have	Extension	✓
		Volumetric Rendering	Should Have	Extension	✓
UI		Camera Controls	Must-Have	Core	✓
		GUI for Parameter Tuning	Must-Have	Core	✓
		Parameter Saving and Loading	Must-Have	Core	✓
		Graphic Design, Animations	Won't Have		
Additional		Procedural Planets	Could Have	Extension	✓

Table 2.1: Project deliverables.

2.5 Software Engineering Tools and Techniques

Development Model I adopted the Agile methodology, emphasizing iterative and incremental development. I broke down each deliverable into smaller tasks and managed them on a Kanban board, organized by development stages. Working in two-week sprints allowed me to ensure consistent progress while remaining flexible in prioritizing important tasks.

Libraries and Language My application was implemented in C++, a popular industry-standard language for computer graphics projects. For shader programming, I chose GLSL (OpenGL Shading Language), integral to OpenGL. Its C-like syntax and comprehensive documentation made it easy to learn and use for shader development. A list of the libraries used is provided in Table 2.2.

Testing Strategy The project's extensive use of GLSL brought unique challenges, notably the scarcity of testing frameworks, debuggers and general functionalities. To counter this, I employed various methods to aid in testing and debugging, detailed in Section 4.2.1.

Import in GLSL To address the lack of import functionality in GLSL, I wrote a custom preprocessor in C++. This allowed me to maintain modular shader files, preventing the codebase from becoming an unwieldy monolith. I included a shader dependency graph in Figure 2.6 to demonstrate the organization of the shader files.

Code Styling I paid careful attention to naming conventions across the C++ and GLSL codebases, enhanced code readability and maintainability. Considering the limited GLSL syntax highlighting in IDEs, I leveraged these naming conventions to implement custom syntax highlighting using regular expressions.

Version Control and Backup For both the code repository and the dissertation write-up, I utilized Git for version control and GitHub as a remote repository for backup. I made sure to make regular commits and pushes.

2.6 Software License

Table 2.2 catalogs the licenses for all the tools and libraries utilized in this project. The **MIT license** was chosen for this project due to its simplicity, permissiveness and compatibility. It allows others to freely use, modify, and distribute the software while providing protection from liability.

2.7 Starting Point

Libraries and Tools My project utilized a selection of pre-existing libraries, as detailed in Table 2.2. Aside from these libraries, the project was developed from scratch.

Prior Experience I had prior experience with OpenGL wrapper libraries, gained through the "Introduction to Graphics" course and a past internship, although my direct experience with OpenGL is limited. I had worked with C++, Dear ImGui and CMake in a previous internship. While I had some exposure to HLSL through "Introduction to Graphics", my experience with GLSL prior to this project was minimal.

Library/Tool	Purpose	License
OpenGL	Establish the core graphics pipeline	Open source for SI
GLFW	Manage desktop application windows	zlib/libpng
Dear ImGui	Develop the application's user interface	MIT
CMake	Maintain a structured and efficient compilation process	BSD
Nlohmann JSON	JSON operations in C++	MIT
STB Image	Image operations in C++	Public Domain or MIT
GLM	Matrix and vector operations in C++	Happy Bunny or MIT

Table 2.2: Overview of purposes and licenses for utilized tools and libraries in the project.

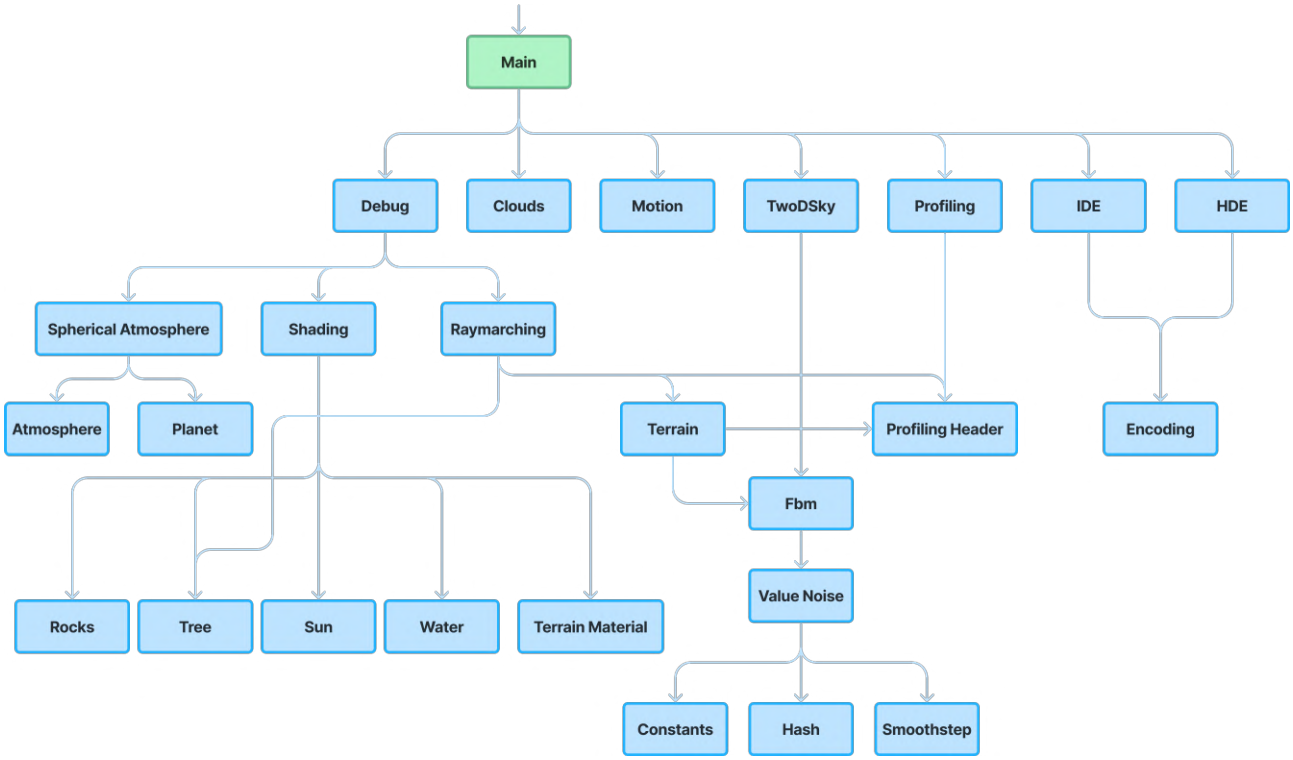


Figure 2.6: Dependency graph of shader files in the project, where A → B denotes "A includes B".

3 Implementation

The application's workflow, as shown in Figure 2.1, consists of three main stages: user interface (UI), procedural generation, and rendering. Given the project's scope—encompassing the procedural generation and rendering of multiple natural elements—it is practical to discuss each element individually, treating UI as a distinct subject.

The chapter begins with an overview of the implementation, followed by detailed sections on each natural element: terrain, trees, water, the atmosphere and clouds. Finally, it discusses procedural planets, input handling, and the UI.

3.1 Implementation Overview

This section outlines the core architecture of the app, focusing on the execution sequences of both the graphics pipeline and the fragment shader within that pipeline.

3.1.1 Pipeline Execution Order

Central to the ray marching rendering approach is the use of a **screen quad**. In this project, since both geometry and rendering processes are encapsulated within the fragment shader, a simple quad that covers the entire screen suffices. This screen quad serves as the canvas for rendering all scene elements. The application pipeline in OpenGL, following the flowchart in Figure 3.1, is implemented in C++ and structured as follows:

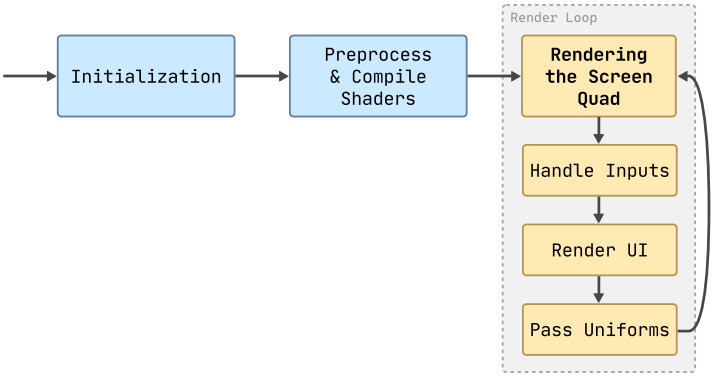


Figure 3.1: Outlines the execution order of the application pipeline.

1. **Initialization** involves creating a window using GLFW, initializing OpenGL, and preparing a Vertex Array Object (VAO) and Vertex Buffer Object (VBO) for the screen quad. The screen quad vertices and indices are predefined constants.
2. **Shader Compilation** includes preprocessing includes and compiling shaders, complete with error reporting.
3. **Render Loop** involves rendering the screen quad, handling user inputs for interactive elements and camera movement, and sending parameters as uniforms to the GPU.

3.1.2 Fragment Shader Execution Order

While procedural generation is a distinct stage in the conceptual workflow, in practice, it occurs **implicitly** within the implementation. The geometry is defined by implicit functions, which are in turn determined by various parameters. Once these parameters are set, the procedural generation phase is effectively complete. Therefore, the focus of the fragment shader's execution order is on the rendering aspects of the scene.

Following the flowchart in Figure 3.2, the fragment shader operates in this sequence:

1. Compute the camera ray and the sun ray.
2. Ray march and render terrain and trees.
3. For pixels with an intersection with terrain or trees, determine the intersection with the water surface and render the water.
4. For pixels without intersections, render the sun disk.
5. Simulate atmospheric Rayleigh scattering for each pixel, resulting in atmospheric perspective effects for intersected pixels, and determining the sky color for pixels without intersections.
6. Render clouds.

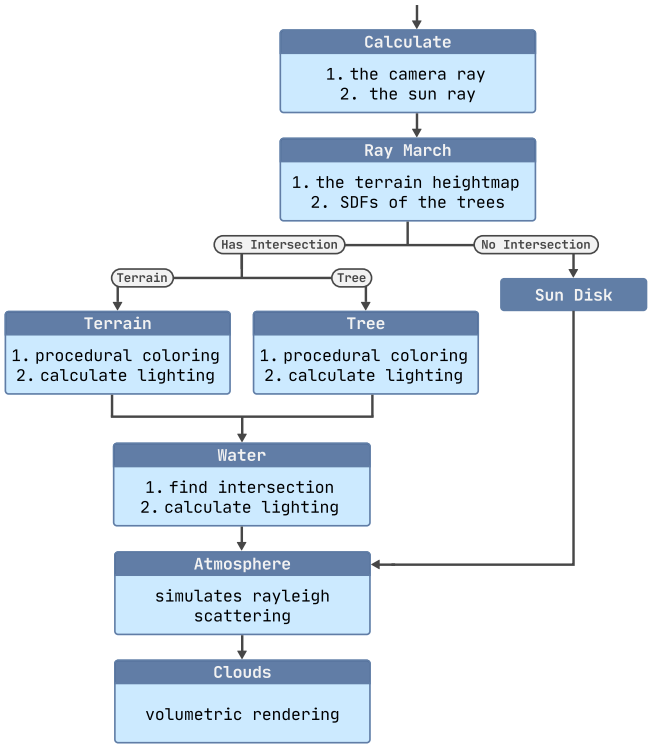


Figure 3.2: Outlines the execution order of the fragment shader.

Details for calculating the camera ray and the sun ray are given in Appendix B.2.

3.2 Terrain

This section details the the implementation for the procedural generation and rendering of terrain in the project.

3.2.1 Procedural Generation

Heightmap Generation

I used the a 2D fBm, fBm_2, to generate a terrain heightmap $H(x, z)$:

$$H(x, z) = v \cdot \text{fBm}_2\left(\frac{(x, z)}{h}\right) \tag{3.1}$$

where v and h are scaling factors, and implementation of fBm is detailed in Appendix C. Parameters for the fBm are made adjustable in the application's UI, as shown in Figure 3.5.

Limitation The generated terrain, despite being infinite, lacks **variety**, typically appearing as uniformly mountainous or plateau-like landscapes, as shown in Figure 3.3.

Dual Heightmap

To make the terrain more varied, I introduce a dual heightmap system:

1. **Local Heightmap** (H_l): Utilizes fBm with higher frequencies to detail terrain on a smaller scale.
2. **Global Heightmap** (H_g): Utilizes fBm with lower frequencies to define the overarching terrain shape and the extent of rockiness.

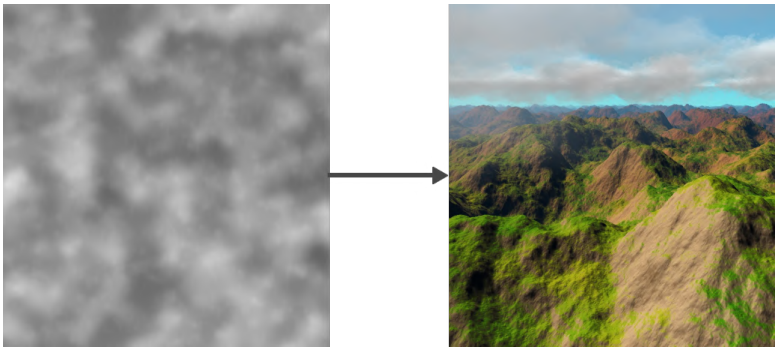


Figure 3.3: *Left*: a heightmap generated from fBm. *Right*: the associated terrain.

The combined heightmap is formulated as follows:

$$H(x, z) = H_g(x, z) + \text{normalize}(H_g(x, z)) \times H_l(x, z) \tag{3.2}$$

The **addition** of $H_g(x, z)$ establishes the general terrain contour, while the **multiplication** by normalize$(H_g(x, z))$ modulates the impact of $H_l(x, z)$, influencing the terrain's rockiness or ruggedness. Lower values of H_g lead to gentler terrain, while higher values induce more pronounced elevation changes, akin to mountainous or rocky landscapes. The dual heightmap approach effectively blends local details with global terrain contours to create diverse and realistic landscapes, as shown in Figure 3.4 and 3.5 for comparison.

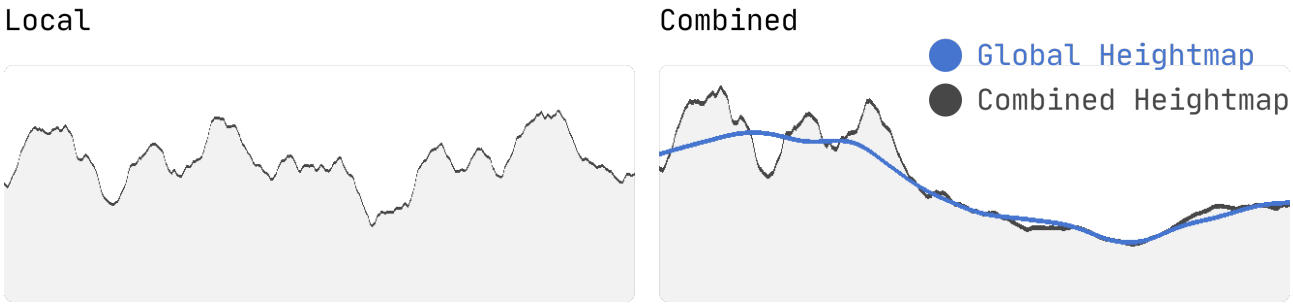


Figure 3.4: Comparison between slices of the local and combined heightmaps along the x-axis.

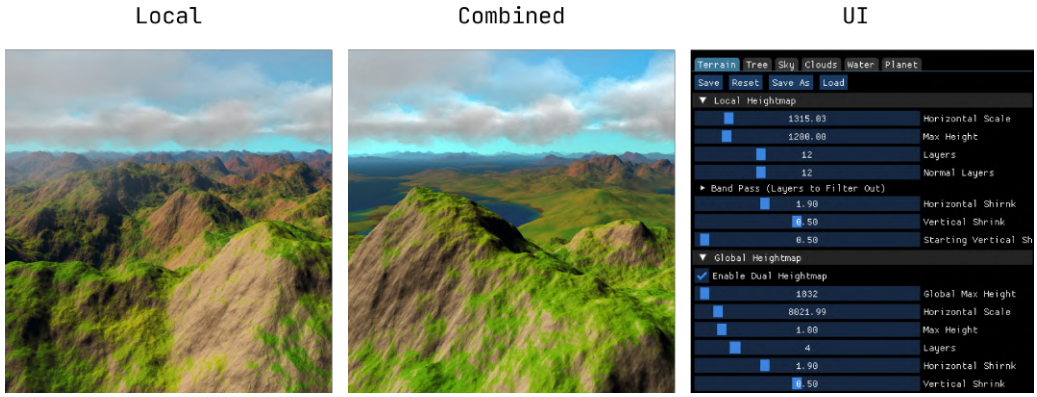


Figure 3.5: The first image shows a mountainous terrain from the local heightmap. Terrain in the second image combines local and global heightmaps, featuring both mountains and plateaus. The third image displays the application's UI with adjustable terrain generation parameters.

Domain Distortion

To create terrain with natural variation, my project applies **domain distortion** to the heightmap. Domain distortion operates by modifying the input domain of a function before evaluation. For a given function $f(x)$, the domain is first distorted by a function $g(x)$, yielding $f(g(x))$ instead of the $f(x)$.

For the heightmap of the terrain, such distortion must be spatially varied to produce interesting results and continuous to avoid abrupt changes. This is achieved by incorporating a secondary 2D fBm, fBm_2', leading to the modified heightmap:

$$H_l(x,z) = v \cdot \text{fBm}_2\left(\frac{x,z}{h} + \gamma \cdot \text{fBm}_2'(x,z)\right) \tag{3.3}$$

where γ is the distortion magnitude, and fBm_2' utilizes distinct parameters. They are all adjustable via the application's UI.

This approach is grounded in the principle that natural landscapes are sculpted by processes such as erosion, sedimentary deposition, and tectonic activities. Domain distortion through fBm_2' gives the generated terrain variations that mimic these geological phenomena. The use of fBm is particularly advantageous as it layers noises of varying frequencies and amplitudes; lower frequencies mimic large-scale geological shifts, while higher frequencies emulate smaller-scale details like erosion. This multi-scale noise application is demonstrated in Figure 3.6.

Choosing suitable parameter values for domain distortion is key. Figure 3.8 displays how varying the distortion magnitude, γ, impacts the heightmap. While fine-tuning these parameters encourages the emergence of natural-looking terrains, pushing the parameters beyond typical ranges can yield intriguing, alien landscapes, as depicted in Figure 3.7.

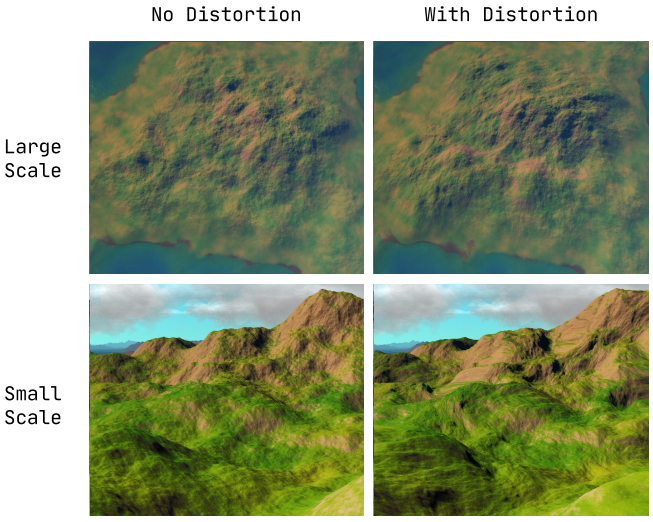


Figure 3.6: Lower frequencies mimic large-scale geological shifts, while higher frequencies emulate smaller-scale details like erosion.

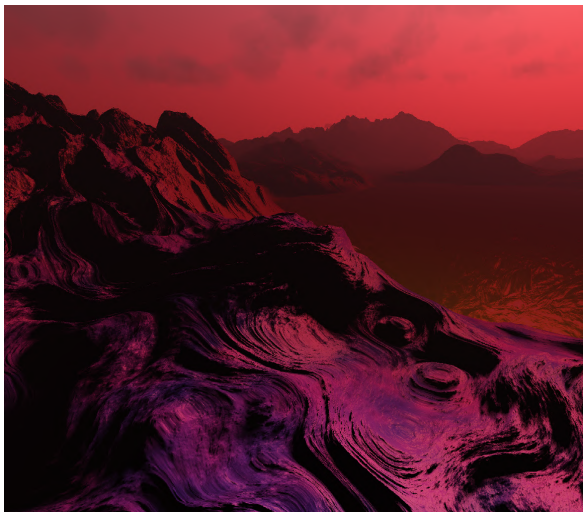


Figure 3.7: Exotic terrain features generated by domain distortion.

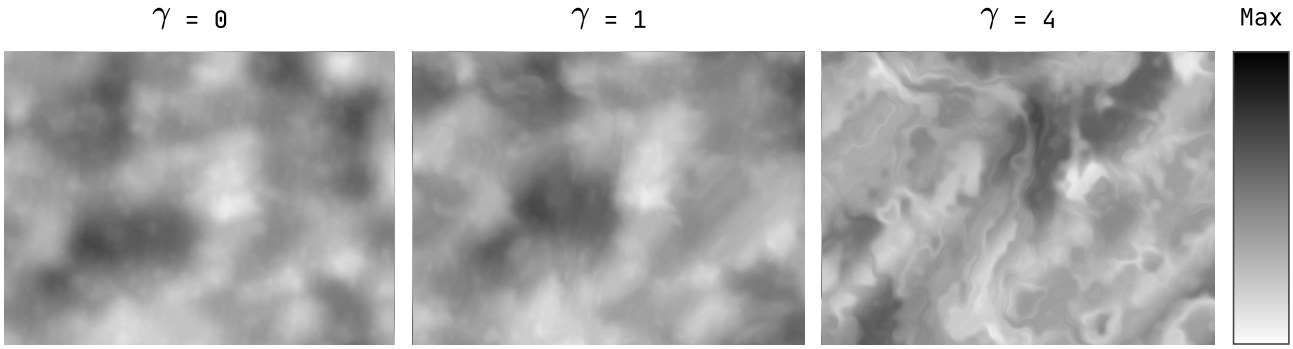


Figure 3.8: Impact of varying distortion magnitude γ on a heightmap.

3.2.2 Ray Marching

In the terrain ray marching process, I optimized the fixed-step ray marching algorithm (Section 2.2.2) by incorporating early termination, dynamic step sizing, and binary search.

Early termination By halting the ray marching process under certain conditions, computational load can be reduced. In my implementation, the process terminates if:

1. The marched distance surpasses a predefined maximum distance, $d_{\max}$.
2. The number of steps exceeds the maximum allowed, denoted as N.
3. A sample point's height exceeds the maximum terrain height (including trees), $y_{\max}$, and the camera ray is ascending ($\mathbf{r_c}.y > 0$). This is based on the understanding that no intersection will occur if the ray is moving away from the terrain.

Dynamic step size The step size in ray marching is dynamically adjusted based on two factors:

1. The step size linearly increases with the **distance** already marched, aligning with the Level of Detail (LOD) concept. Farther objects can have less detail without significantly impacting the visual quality.
2. The step size also scales with the **height** above the terrain. This heuristic allows faster traversal of areas far from the terrain.

The step size for the ith step, s_i, is defined as:

$$s_i = a + b \cdot d_i + c \cdot (\mathbf{P}_i.y - H(\mathbf{P}_i.xz)) \tag{3.4}$$

where a, b and c are adjustable parameters, and d_i is the distance marched to the ith point.

The final stage in the ray marching process involves refining the intersection point for enhanced precision. Traditionally (Section 2.2.2), this is done through linear interpolation between the last two points. In my approach, I have improved this process by employing a binary search method, allowing for a more precise intersection determination, as illustrated in Figure 3.9.

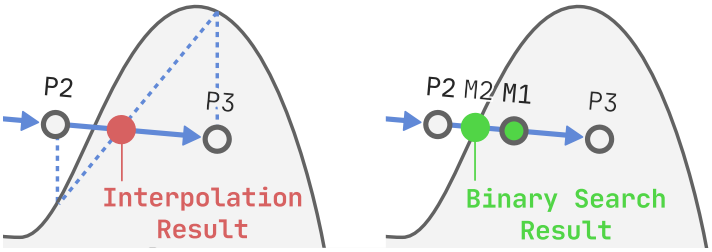


Figure 3.9: Comparing linear interpolation and binary search for terrain intersections. The right diagram shows a 2-step binary search, where $\mathbf{M}_1$ is below the terrain, leading to $\mathbf{M}_2$ being the midpoint between $\mathbf{P}_2$ and $\mathbf{M}_1$.

3.2.3 Procedural Coloring

In my project, the terrain is categorized into **four distinct elevation-based regions**: mud (below water level), sand, grass, and rock, as illustrated in Figure 3.10.

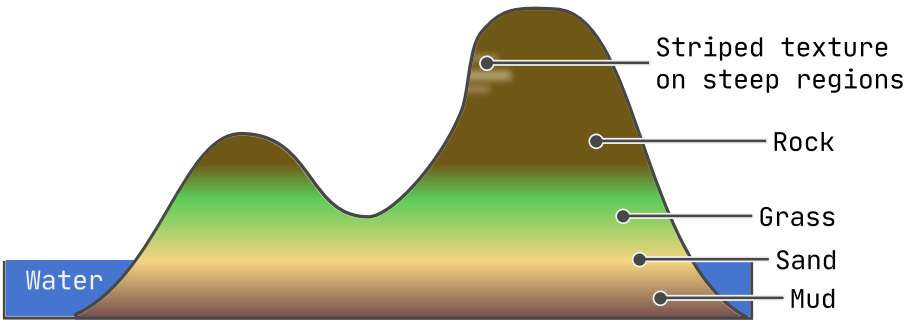


Figure 3.10: Cross-sectional representation of terrain illustrating the four elevation-based regions. Striped textures are applied to steep areas.

Smooth Boundaries The elevation thresholds separating these regions are denoted as $h_1 < h_2 < h_3$. Rather than using rigid boundaries, each threshold incorporates a variation $\pm\delta_i$ to facilitate smooth transitions. This is achieved by using the smoothstep function to interpolate between regions.

Dynamic Boundaries To enhance realism and avoid monotonous separation at fixed elevations, a 2D fBm function offsets these boundaries, as demonstrated in Figure 3.11. Consequently, at any point $\mathbf{P}$, the modified boundary is expressed as:

$$h_i + \text{fBm}_2(\mathbf{P}.xz) \pm \delta_i \tag{3.5}$$

The Grass Region Grass coverage is determined by the slope's steepness: grass is rendered only on surfaces where the normal's y-component is below a predefined threshold, indicating flatter regions. Interpolation is used to ensure smooth transition. Additionally, I introduced **color variations** by evaluating another 2D fBm on the point's xz coordinates. By smoothly transitioning between two distinct grass colors through interpolation, this technique offers a more dynamic and visually appealing depiction of grassy areas, as shown in Figure 3.12.

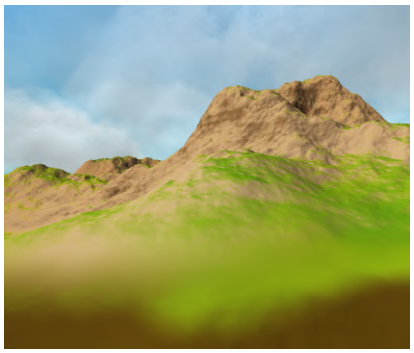


Figure 3.11: Render showing smooth, dynamic boundary adjustments between regions.

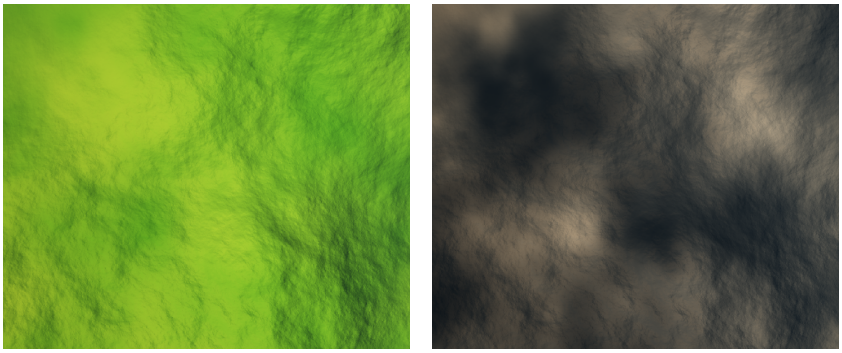


Figure 3.12: Aerial view of the color-varied grass region. *Left*: terrain with natural grass color variation, *Right*: emphasizes color variations in high-contrast black and white.

Striped Rock Texture In steep areas, I have procedurally generated a rock texture resembling striped patterns found on mountains, as illustrated in Figure 3.13. This effect is created by horizontally stretching value noise, producing stripe-like but non-uniform patterns. The degree of texture visibility is controlled by interpolating between the textured appearance and the rock's base color using the y-component of the normal. This ensures that stripes are prominent in steep regions, with a seamless transition to standard rock texture on flatter surfaces. Considering that the texture only appears on steep regions, I used the point's xy components as UV coordinates for texture sampling. **Importantly**, the texturing process does not involve creating and sampling from an image; rather, it queries a function directly.

3.2.4 Illumination and Shadows

Shading In shading the terrain, I utilized the Phong reflection model [24], complemented by Fresnel Reflection (Section 2.2.3). The Phong model requires several inputs: the intersection point determined through ray marching (Section 3.2.2), the material color via procedural coloring (Section 3.2.3), the incident light direction from the sun's spherical coordinates (Appendix B.2.2), and the terrain normal, analytically derived from the fBm (Appendix C).

For added realism, the Phong shading model is augmented with Fresnel Reflection. Using Schlick's approximation (Equation 2.9) to compute the Fresnel term, $F(\theta)$, the specular term within the Phong model, I_s, is modified to be:

$$I_s = k_s \cdot F(\theta) \cdot (\mathbf{r} \cdot -\mathbf{cr})^n \cdot I \tag{3.6}$$

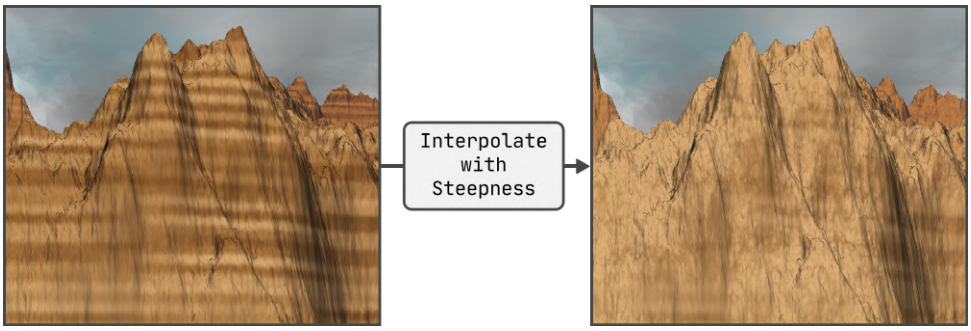


Figure 3.13: *Left*: the application of a striped pattern texture across the terrain. *Right*: the effect of interpolating this texture with the steepness of the terrain, resulting in the striped pattern showing only on steep slopes.

Shadows To make terrain cast soft shadows (as demonstrated in Figure 3.15), I implemented a method for calculating the penumbra factor, P, based on the approach in Section 2.2.3. Instead of measuring the actual distance from a shadow sample point to the terrain, I used vertical distance as a more efficient approximation. This method is visualized in Figure 3.14 and implemented as outlined in Algorithm 1.

The shadow intensity, S, is subsequently integrated into the terrain's lighting model by multiplying with the final color $\mathbf{c}$ of the fragment.

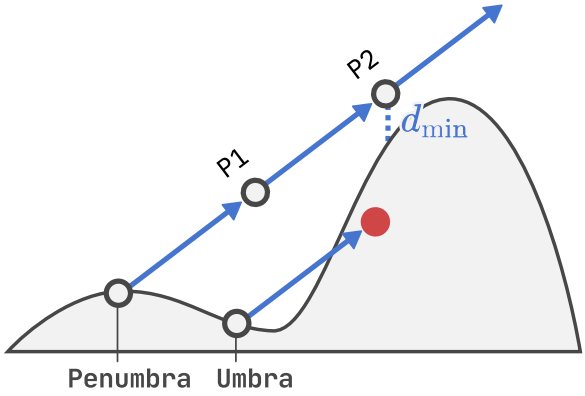
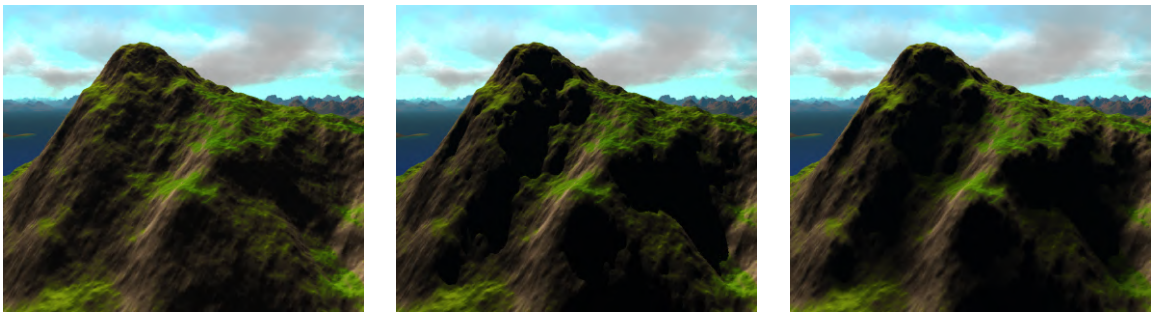


Figure 3.14: Soft shadow calculations for terrains. For the point in penumbra, its shadow ray doesn't intersect with the terrain, with $d_{\min}$ incurred at $\mathbf{P}_2$.

Algorithm 1 Terrain Shadow Calculation

function TERRAIN_SHADOW($\mathbf{X}, \mathbf{r_s}$)
  $d_{\min} \leftarrow \infty$
  for $i = 0$ to $s_{\max}$ **do**
    $\mathbf{P}_i \leftarrow \mathbf{X} + i \cdot s \cdot \mathbf{r_s}$
    $d_i \leftarrow \mathbf{P}_i.y - H(\mathbf{P}_i.xz)$
    $d_{\min} \leftarrow \min(d_{\min}, \frac{d_i}{i \cdot s})$
  end for
  return smoothstep$(0, 1, d_{\min})$
end function



(a) No Shadows (b) Hard Shadows (c) Soft Shadows

Figure 3.15: Comparison of terrain shadows.

3.3 Trees

This section details the the implementation for the procedural generation and rendering of trees in the project.

3.3.1 Procedural Generation

In my project, each tree is modeled using the SDF (Section 2.1.3) of an ellipsoid, with added 3D fBm to mimic the foliage, and populated using domain repetition.

SDF of Ellipsoids Instead of a simply scaling a sphere's SDF, I used a more accurate representation for ellipsoids [28]:

$$\mathrm{SDF}_{\mathrm{ellipsoid}}(\mathbf{X}, \mathbf{r}) = \frac{\left\|\frac{\mathbf{X}}{\mathbf{r}}\right\| \cdot \left(\left\|\frac{\mathbf{X}}{\mathbf{r}}\right\| - 1\right)}{\left\|\frac{\mathbf{X}}{\mathbf{r}\cdot\mathbf{r}}\right\|} \tag{3.7}$$

fBm for Leaves The leaves are simulated by distorting the ellipsoid with a 3D fBm:

$$\mathrm{SDF}_{\mathrm{tree}}(\mathbf{X}) = \mathrm{SDF}_{\mathrm{ellipsoid}}(\mathbf{X}, \mathbf{r}) + \gamma \cdot \mathrm{fBm}_3(\mathbf{X}) \tag{3.8}$$

Here, γ represents the distortion magnitude, and it is adjustable in the application's UI along with the fBm parameters. The dimension $\mathbf{r}$ of the ellipsoids is randomized within a range to introduce size variation among trees.

Domain repetition for tree population To efficiently distribute trees across the terrain, a domain repetition [29] technique is used. The SDF for a tree instance is made periodic on an infinite axis-aligned 2D square grid in the xz-plane. This results in a representation of infinite trees within this grid pattern, as illustrated in Figure 3.16. The tree positions are vertically adjusted by the terrain height and an additional constant v:

$$\begin{aligned} \mathrm{SDF}_{\mathrm{trees}}(\mathbf{X}) &= \mathrm{SDF}_{\mathrm{tree}}(\mathbf{X} - \mathbf{C}), \\ \text{where } \mathbf{C}.x = \mathbf{C}.z &= \left(\left\lfloor \frac{\mathbf{X}}{d} \right\rfloor + 0.5\right) \cdot d + o(\mathbf{X}), \\ \mathbf{C}.y &= H(\mathbf{X}.xz) + v. \end{aligned} \tag{3.9}$$

Here, $\mathbf{C}$ represents the center of the cell that $\mathbf{X}$ is in, d represents the cell size in the square grid, and $o(\mathbf{X})$ is a random offset based on the position hashed. Moreover, trees are placed only on terrain that meets two criteria: the slope must be below a certain threshold to avoid cliffs, and the elevation must be higher than the water surface to prevent trees in water. The randomization of offsets and dimensions are depicted in Figure 3.16.

Considering Neighboring Cells The random offset $o(\mathbf{X})$ introduces complexity in determining the closest tree to any given point $\mathbf{X}$, as it may not necessarily be within the same cell as $\mathbf{X}$. To address this, I evaluate the surrounding eight cells, along with the cell containing $\mathbf{X}$, to find the nearest tree. The SDF for the trees, therefore, is calculated by finding the minimum distance to a tree among these nine cells:

$$\begin{aligned} \mathrm{SDF}_{\mathrm{trees}}(\mathbf{X}) &= \min_{i\in\{-1,0,1\}} \min_{j\in\{-1,0,1\}} \mathrm{SDF}_{\mathrm{tree}}(\mathbf{X} - \mathbf{C_{ij}}), \\ \text{where } \mathbf{C_{ij}}.x &= \left(\left\lfloor \frac{\mathbf{X}}{d} \right\rfloor + i + 0.5\right) \cdot d + o(\mathbf{X}), \\ \mathbf{C_{ij}}.z &= \left(\left\lfloor \frac{\mathbf{X}}{d} \right\rfloor + j + 0.5\right) \cdot d + o(\mathbf{X}), \\ \mathbf{C_{ij}}.y &= H(\mathbf{X}.xz) + v. \end{aligned} \tag{3.10}$$

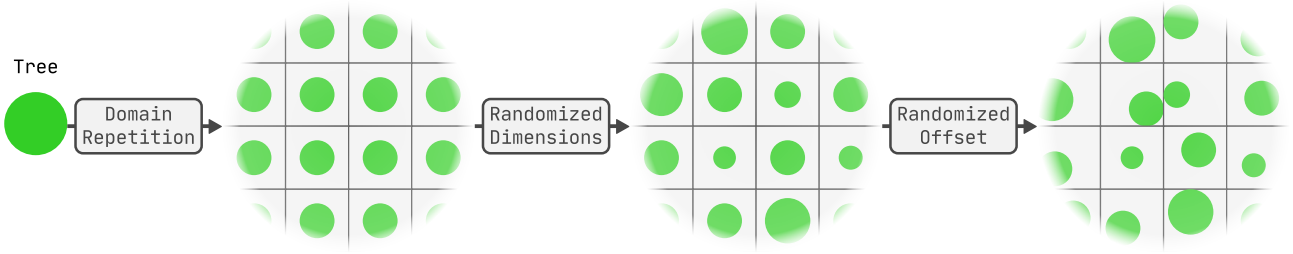


Figure 3.16: Depicts domain repetition and the incorporation of randomized offsets and dimensions in tree generation, excluding species variation for simplicity.

3.3.2 Ray Marching

While Sphere Tracing (Section 2.2.1) is a viable method for determining the intersection with the SDF of trees, initiating this process from the camera position would be inefficient, considering that a similar ray marching process has just been done for the terrain. To optimize efficiency, I integrated checks for intersections with **tree canopies** into the terrain ray marching algorithm. The tree canopy, defined by a constant height above the terrain, serves as an approximated bounding box for the trees, as depicted in Figure 3.17. Once the terrain intersection is identified through ray marching, the farthest canopy intersection point is then used as the starting position for Sphere Tracing of the trees. This approach substantially reduces sphere tracing steps, as illustrated in Figure 3.17.

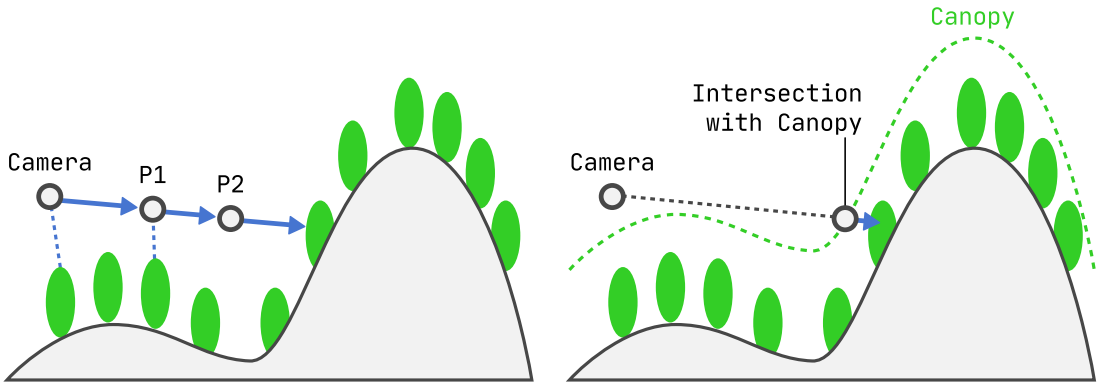


Figure 3.17: Illustrating the efficiency of tree ray marching: Sphere Tracing from the camera (left) takes 3 steps, while starting from the canopy intersection (right) requires only 1 step.

3.3.3 Procedural Coloring

In the procedural coloring of trees, the primary objective is to introduce **color variation** among trees. The resulting variations are shown in Figure 3.18. I implemented two types of color variations: inter-species and intra-species. Each generated tree belongs to either species A or B (determined by a 2D fBm), leading to the first type of variation, where different color schemes distinguish between the two species. Within each species, I simulated color variation due to age differences by generating a random value representing a tree's age. This value is used to interpolate between the 'young' and 'old' colors specific to the species.

3.3.4 Illumination and Shadows

Shading For effective lighting of trees, determining the normal at the intersection point $\mathbf{P}$ is crucial. First, I obtain the normal of the tree's SDF using a numerical method (Appendix B.3). Subsequently, this SDF normal is blended with the normal of the underlying terrain at point $\mathbf{P}$. This blending facilitates the representation of tree lighting that conforms to the shape of

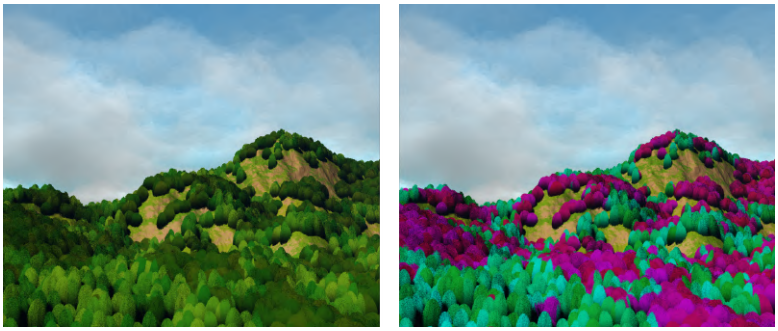


Figure 3.18: *Left*: trees of one species in more saturated colors and another species in less saturated colors, with color variations indicating age differences. *Rigth*: uses higher contrast colors to differentiate the two species.

the terrain beneath, expressed as:

$$\mathbf{n}(\mathbf{P}) = a \cdot \mathbf{n}_{\text{SDF}}(\mathbf{P}) + (1 - a) \cdot \mathbf{n}_H(\mathbf{P}.xz) \tag{3.11}$$

Here, $\mathbf{n}_{\text{SDF}}(\mathbf{P})$ is the SDF normal at point $\mathbf{P}$, $\mathbf{n}_H$ is the normal of the underlying terrain, and a is a blending factor.

Similar to the shading approach for terrain (Section 3.2.4), the Phong Model with Fresnel reflection is employed for shading trees. Additionally, to approximate self-occlusion and ambient occlusion by nearby trees, I introduce an occlusion term $O = \text{smoothstep}(a, b, h_t)$, interpolated between 0 and 1 based on the height of the tree h_t, with parameters a and b controlling the interpolation. The diffuse component of the lighting model is then multiplied with O, resulting in lower parts of the tree appearing less lit than the upper parts, as demonstrated in Figure 3.19.

Shadows To make trees cast soft shadows, I adapted the terrain shadowing technique in Section 3.2.4 to accommodate the semi-transparent nature of tree foliage. Unlike the solid terrain, foliage partially allows light to pass through. I modeled this by evaluating how deeply a shadow ray penetrates the tree's volume—deeper penetration near the trunk indicates darker shadows.

Following Algorithm 2, I calculated the shadow intensity by accumulating the value $(\text{SDF}_{\text{trees}}(\mathbf{P_s}) - \lambda) \cdot t$ along the shadow ray. Here, $\mathbf{P_s}$ denotes points along the ray, t the ray's distance, and λ a threshold for the SDF in shadow calculations. The accumulated value, D, is interpolated using a smoothstep function to determine the shadow intensity, S. A low D value (below D_{min}) indicates that the point is in full shadow (umbra). The effects are illustrated in Figure 3.19.

Algorithm 2 Tree Shadow

```
function TreeShadow(P)
    D ← 0
    for t = 0 to t_max step s do
        P_s ← P + t · r_s
        d ← tree_sdf(P_s)
        if d < λ then
            D ← D + (d − λ) · t
        end if
        if D ≤ D_min then
            break
        end if
    end for
    S ← smoothstep(D_min, 0, D)
    return S
end function
```

3.3.5 Core Version Completed

Following the implementation of trees, I extended the application with simple, basic implementations of the clouds and the sky (Appendix D), marking the completion of the core version of the application. Please refer to Appendix A.1 for renders from this version.

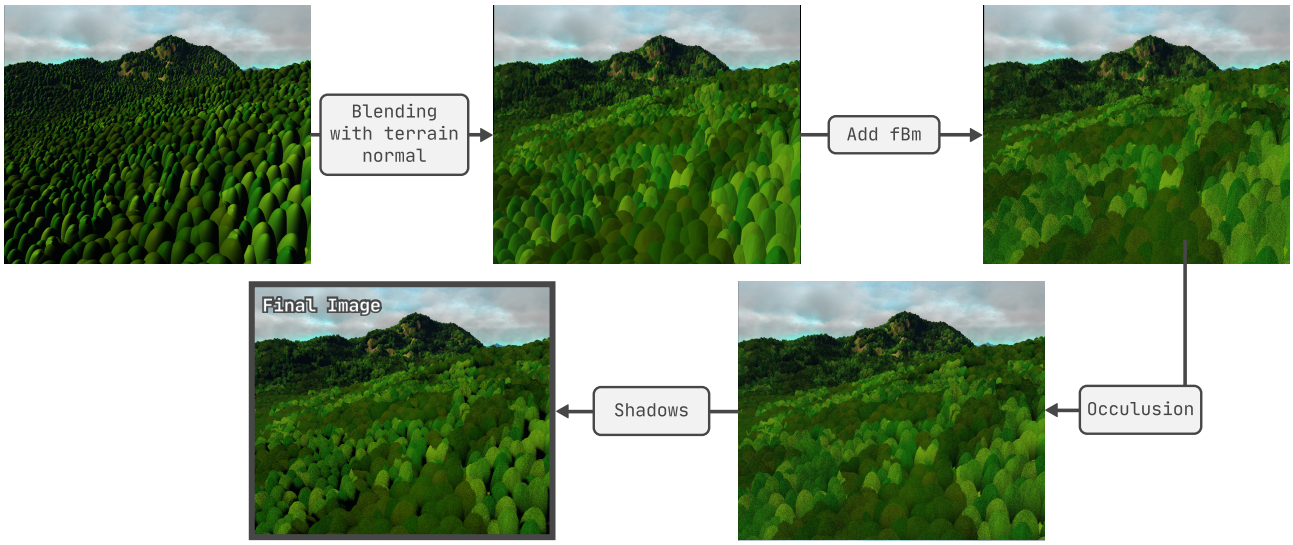


Figure 3.19: Illustrating the complete process of tree illumination and shadowing.

3.4 Extension: Water

This section details the my implementation for the representation and rendering of water.

3.4.1 Representation

This project used a simple model to represent water. Assuming a constant water level globally, the water surface is modeled as a plane perpendicular to the y-axis. The elevation of this surface, denoted as w, can be adjusted in the UI. Instead of geometrically distorting the water surface to simulate waves, normal mapping [30] was applied to mimic wave effects, as detailed in Section 3.4.4.

3.4.2 Finding Intersection

In the shader execution order (Section 3.1.2), the intersection with the water is determined **after** the identification of terrain and tree intersections. This order is chosen because water is transparent, and its visibility depends on whether it is occluded by these elements. If the intersection point with either a tree or terrain is beneath the water level w, the water is considered visible. Then the intersection point with the water surface, $\mathbf{P}_w$, and the distance d_w from the camera $\mathbf{C}$ to the water surface are determined by:

$$\mathbf{P}_w = \mathbf{C} + d_w \cdot \mathbf{r_c}, \;\; d_w = \frac{w - \mathbf{C}.y}{\mathbf{r_c}.y} \tag{3.12}$$

3.4.3 Procedural Coloring

The procedural coloring of water focuses on depth-dependent color variation and transparency.

Depth-Based Color Variation The effective length of water that the camera ray travels through, denoted as l, is computed as the difference between the total distance to an object d_o and the distance to the water surface d_w. The **transmittance** of water, τ, quantifies the amount of light that passes through the water. It decreases exponentially with the increase in l, modulated by a decay factor k: $\tau = \exp(-k \times l)$. Water color is then interpolated between deep and shallow hues based on this transmittance τ, emulating how water absorbs and scatters light differently at varying depths.

Transparency Effect Transparency is achieved by blending the water material color with the color at the underwater terrain intersection, based on the transmittance τ. A higher τ indicates more light scatter, resulting in less light penetration and decreased transparency.

3.4.4 Illumination

A heightmap $H(x, z)$, constructed from a 2D fBm, represents water surface waves. The normal of this heightmap, N_H, serves as the normal map. Shading follows similar techniques as for terrain, utilizing the Phong model with a Fresnel effect, as detailed in Section 3.2.4.

3.4.5 Example Output

Please refer to Appendix A.2 for example renders of water scenes, and to `water_in_motion.mp4` to the waves in motion.

3.5 Extension: Atmosphere

This section outlines the physics-based rendering techniques implemented for simulating atmospheric effects, primarily focusing on Rayleigh scattering [31]. This phenomenon, caused by the interaction of sunlight with minuscule atmospheric particles, is crucial for achieving authentic sky colors and atmospheric perspective, as demonstrated in renders in Figure A.3 and A.4 respectively.

The simulation examines the path of light from a point $\mathbf{E}$ in the scene to the camera at position $\mathbf{C}$, counter to the direction of the camera ray $\mathbf{r_c}$. It accounts for two key phenomena:

- **In-scattering**: Light scatters from other directions into the direction of the ray.
- **Out-scattering**: The loss of light from the ray as it scatters in other directions.

3.5.1 Modeling the Atmosphere

Since the procedurally generated world is infinite in the xz direction, the atmospheric density depends solely on altitude. The atmosphere is given an upper limit $h_{\max}$, above which the density is zero. The atmospheric density D at a point $\mathbf{P}$ below the upper bound is modeled as:

$$D(\mathbf{P}) = \frac{e^{-\lambda \mathbf{P}.y} \left(h_{\max} - \mathbf{P}.y\right)}{h_{\max}} \tag{3.13}$$

where λ is a decay constant modifiable in the UI.

3.5.2 The In-Scattering Equation

The in-scattering equation [18] models the accumulation of sunlight along a ray from point $\mathbf{E}$ to the camera $\mathbf{C}$ through the atmosphere. It encompasses both in-scattering and out-scattering. Figure 3.20 provides a visual aid for understanding the related notations. The equation is expressed as:

$$I_v(\lambda) = I(\lambda) \times K(\lambda) \times F(\theta, g) \times \int_{\mathbf{C}}^{\mathbf{E}} \left(D(\mathbf{P}) \times \exp\left(-t\left(\mathbf{PS}, \lambda\right) - t\left(\mathbf{PC}, \lambda\right)\right)\right) ds \tag{3.14}$$

where $I(\lambda)$ denotes the intensity of sunlight at wavelength λ. $K(\lambda)$ is the wavelength-dependent scattering coefficient. In the implementation, the coefficients for the red, green, and blue wavelengths are used, with their values being adjustable in the UI. $F(\theta, g)$ is the phase function, representing the angular distribution of scattering.

The end point $\mathbf{E}$ varies based on scene interaction. If the camera ray intersects with an object (terrain, tree, or water), $\mathbf{E}$ is the intersection point, as shown in the right diagram of Figure

3.20. If there's no intersection, $\mathbf{E}$ is the point where the camera ray $\mathbf{r_c}$ intersects the upper boundary of the atmosphere, as shown in the left diagram of Figure 3.20.

In the integral, $\mathbf{P}$ is a point on the segment between $\mathbf{C}$ and $\mathbf{E}$. $P = \mathbf{R}(s) = \mathbf{C} + s \cdot \mathbf{r_c}$, where $\mathbf{R}$ is the camera ray. t is the out-scattering function, H_0 is a parameter adjustable in the UI, and $\mathbf{S}$ is the intersection between the atmosphere's upper boundary with the ray with origin $\mathbf{P}$ and direction $\mathbf{r_s}$, as depicted in Figure 3.20.

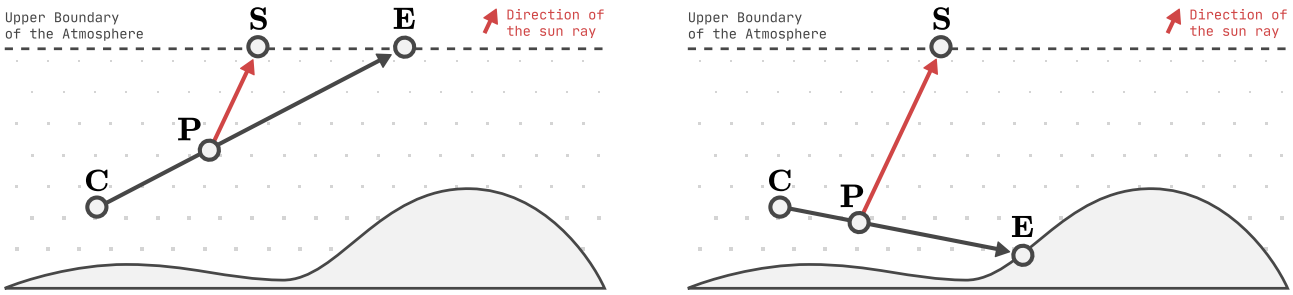


Figure 3.20: Visual clarification of the in-scattering equation notations and concepts. *Left*: No intersection, *Right*: Intersection Present.

The integral is approximated using Riemann sums [32], a method of summing rectangular areas to approximate the area under a curve. This technique involves marching along the camera ray, sampling at points $\mathbf{P}_i$ and evaluating the integrand at each point, then multiplying by the step size s. The sum of these calculations approximates the integral in the scattering equation. This process is illustrated in Figure 3.21.

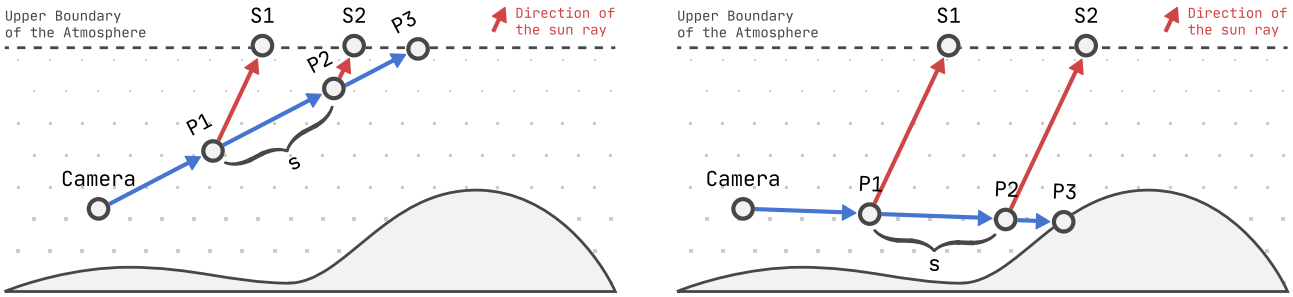


Figure 3.21: Visually demonstrates the Riemann sums method for the in-scattering equation. *Left*: No intersection, *Right*: Intersection Present.

The Out-Scattering Function

The out-scattering function t describes the reduction of light as it traverses through the atmosphere and is given by:

$$t(\mathbf{A}, \mathbf{B}, \lambda) = 4\pi \times K(\lambda) \times \int_{\mathbf{A}}^{\mathbf{B}} D(\mathbf{P}) ds \tag{3.15}$$

Optical Depth The integral represents the optical depth, essentially the atmospheric density averaged along the path segment multiplied by the length of the segment. This value indicates the cumulative effect of atmospheric particles on light along the ray path: more particles imply greater scattering of light away from the ray. To approximate the optical depth, the Riemann sum method was employed, similar to the approach outlined in Section 3.5.2.

The Phase Function The phase function F, essential in determining the directionality of scattering, is adapted from the Henyey-Greenstein function [18]:

$$F(\theta, g) = \frac{3 \times (1 - g^2)}{2 \times (2 + g^2)} \times \frac{1 + \cos^2 \theta}{(1 + g^2 - 2 \times g \times \cos \theta)^{\frac{3}{2}}} \tag{3.16}$$

Here, θ is the angle between the incoming light and the scattering direction. The asymmetry parameter g influences the scattering distribution. For Rayleigh scattering, where $g = 0$, the function simplifies to $\frac{3}{4}(1 + \cos^2\theta)$, indicating more light scattered in directions close to the incoming light source.

3.5.3 Blending

The blending process involves combining the in-scattered light with light emitted at intersections in the scene. When the camera ray doesn't intersect any scene object, the sky color is determined solely by in-scattered light. However, at intersection points, the blending of in-scattered and emitted light is necessary, given by [18]:

$$I'_v(\lambda) = I_v(\lambda) + I_e(\lambda)\exp(-t(\mathbf{C}, \mathbf{E}, \lambda)) \tag{3.17}$$

where $I_v(\lambda)$ represents the in-scattered light at wavelength λ, and $I_e(\lambda)$ is the light emitted at the intersection $\mathbf{E}$. For the final blended color $\mathbf{c}$ with RGB representation, I focus on three specific wavelengths corresponding to red, green, and blue light:

$$\mathbf{c} = (I'_v(\lambda_r), I'_v(\lambda_g), I'_v(\lambda_b)) \tag{3.18}$$

3.5.4 Example Output

Please refer to Appendix A.3 for example renders of the physics-simulated atmosphere, and to `interactive_atmosphere.mp4` to observe real-time changes in the atmosphere with varying sun directions.

3.6 Extension: Clouds

3.6.1 Procedural Generation of Clouds

In modeling clouds, a **volumetric** approach is essential due to their complex, three-dimensional structure. Unlike terrain, clouds are not just surfaces; they are composed of particles with varying densities in three-dimensional space. Therefore I used a density map, $D(\mathbf{X})$, to represent the density at any given point $\mathbf{X}$ in 3D space.

The procedural generation of this map begins with establishing a **bounding box** for the clouds, extending infinitely in the xz-plane and bounded vertically between lower y_l and upper y_u limits. The generation involves two primary steps (as illustrated in Figure 3.22):

1. The **base density** of the clouds is initially determined relative to their vertical position. The cloud density is intensified nearer to the box's central vertical axis, denoted by y_c, the mean of y_u and y_l. The base density at a point $\mathbf{x}$ is formulated as:

   $$D(\mathbf{X}) = h - |\mathbf{X}.y - y_c| \tag{3.19}$$

   where h is a fraction of the total vertical extent of the cloud box.
2. To this base density, a **3D fBm** is added, introducing natural, cloud-like irregularities and variations. The final density $D(\mathbf{x})$ at point $\mathbf{x}$ is then:

   $$D(\mathbf{X}) = h - |\mathbf{X}.y - y_c| + \text{fBm}_3(\mathbf{X}) \tag{3.20}$$

In this approach, the combination of a base density gradient and the nature-simulating properties of fBm (Section 2.1.2) results in a realistic representation of cloud density. This method ensures clouds exhibit natural variations, while predominantly clustering towards the bounding box's center, avoiding unrealistic overflow.

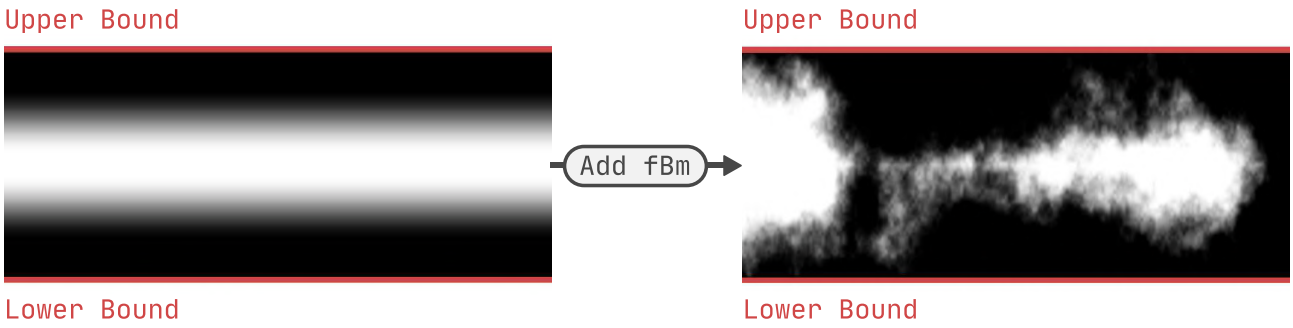


Figure 3.22: Illustrates the procedural generation process of the cloud density map. The diagram displays cross-sectional slices of the density map along the x-axis.

3.6.2 Volumetric Rendering

Due to their transparent and volumetric properties, clouds cannot be rendered just using traditional ray marching to locate surface intersections. Instead, my approach involves sampling multiple points along the camera ray within the cloud volume. At each sampled point, color is calculated, and these colors are subsequently blended to form the final color.

Volumetric Ray Marching

The volumetric ray marching process involves calculating the start and end points of the ray within the cloud's bounding box and then marching the ray through the volume.

Start and end points The start t_s and end t_e distances along the ray from camera position $\mathbf{C}$ are determined by intersecting the ray with the cloud's vertical boundaries, y_l and y_u:

$$t_s = \max(\min(\frac{y_l - \mathbf{C}.y}{\mathbf{r_c}.y}, \frac{y_u - \mathbf{C}.y}{\mathbf{r_c}.y}), 0), \;\; t_e = \min(\max(\frac{y_l - \mathbf{C}.y}{\mathbf{r_c}.y}, \frac{y_u - \mathbf{C}.y}{\mathbf{r_c}.y}), t_{\max}) \tag{3.21}$$

Here, $t_{\max}$ is the maximum ray marching distance. The computations consider all the cases: where the camera position is below, in or above the bounding box.

Adaptive step size The step size s is dynamically adjusted based on the cloud density $D(\mathbf{X})$ and the distance t already covered by the ray. The density function $D(\mathbf{X})$ includes negative values, which, while uncharacteristic for physical densities, are utilized here for computational efficiency. These negative values indicate regions devoid of clouds, allowing the algorithm to increase the step size s in proportion to the negativity of the density, thereby speeding up the march through empty spaces.

For positive density regions, the step size is increased linearly with distance t to reduce computational load, as distant samples contribute less to visual detail and the final color blend.

Termination criteria Ray marching is terminated under one of three conditions:

1. The ray exits the bounding box ($t \geq t_e$).
2. A maximum number of steps is reached.
3. The accumulated alpha value of the blended color reaches a threshold indicating sufficient opacity (as detailed in Section 3.6.2).

Illumination and Shadows

Lighting is calculated at every sample point in the volumetric rendering process. The Phong model [24] is adapted here by excluding the specular term, since the primary visual characteristics of particles in clouds are diffuse scattering and translucency.

Shadows within the clouds, resulting from cloud particles obstructing light from other particles, are rendered using a technique called volumetric shadowing. This process involves marching a ray along the direction of the sun ray and accumulating the density of cloud particles at various points along this path. Higher densities indicate a greater number of particles blocking the light, resulting in deeper shadows at the sample point, and vice versa.

Blending

The blending is done cumulatively, starting from the nearest point to the viewer and progressing towards the back. A cumulative color vector $\mathbf{c}$ (with RGBA components) is maintained and updated along the ray path. For each sampled point $\mathbf{X_i}$, the alpha value α is calculated based on the density $D(\mathbf{X}_i)$ at that point, the step size s, and a scaling factor λ, with an upper limit set by the maximum cumulative alpha $\alpha_{\max}$:

$$\alpha = \text{clamp}(\lambda \times s \times D(\mathbf{X}_i), 0, \alpha_{\max}) \tag{3.22}$$

The cumulative color $\mathbf{c}$ is then blended with the color $\mathbf{c}_i$ at each sampled point $\mathbf{X_i}$:

$$\mathbf{c} \mathrel{+}= \mathbf{c}_i \times \alpha \times (\alpha_{\max} - \mathbf{c}.\alpha) \tag{3.23}$$

Here, α prioritizes points with higher density, while the factor $(\alpha_{max} - \mathbf{c}.\alpha)$ prioritizes closer points. Finally, the cloud color is blended with the fragment color using $\mathbf{c}.\alpha$.

Sunlight Penetration Effect

An additional effect emulates sunlight filtering through clouds, especially noticeable when the sun is behind them, as demonstrated in Figure 3.23. This is achieved by augmenting $\mathbf{c}$ with the sun's color $\mathbf{c}_{\text{sun}}$, modulated by $\mathbf{c}.\alpha$, and the angle between the camera ray direction $\mathbf{r_c}$ and the sun direction $\mathbf{r_s}$:

$$\mathbf{c}.rgb \mathrel{+}= a \cdot \mathbf{c}_{\text{sun}} \cdot (1 - b \cdot \mathbf{c}.\alpha) \cdot (\text{clamp}(\mathbf{r_c} \cdot \mathbf{r_s}, 0, 1))^c \tag{3.24}$$

Here, a, b, c are parameters that are adjustable in the UI. $\mathbf{c}.\alpha$ indicates the cumulative cloud density along the camera ray, with a lower value indicating less cloud density and therefore more sunlight filtering through. Moreover, the effect is stronger when the camera ray is closely aligned with the sun ray, indicated by $\mathbf{r_c} \cdot \mathbf{r_s}$.

(a) Effect off. (b) Effect on.

Figure 3.23: Comparison of cloud renders demonstrating sunlight penetration effect.

3.6.3 Example Output

Please refer to Appendix A.4 for example renders of volumetric clouds, and to `clouds_in_motion.mp4` to see them in motion.

3.7 Extension: Procedural Planets

Since development was on schedule, I had time to implement an extension: applying the existing procedural generation and rendering of natural elements to planets. This section explains how I generate planet heightmaps, render the planet surface, and incorporate water and atmosphere.

3.7.1 Heightmap Generation

In my project, the heightmaps of planets differ from those of flat terrains. For a planet centered at point **O**, given a direction **OP**, the point **I** is where the ray in the direction of **OP** intersects the planet's surface. The heightmap function $H_{\text{planet}}(\mathbf{P})$ returns the length $\|\mathbf{OI}\|$, as illustrated in Figure 3.24.

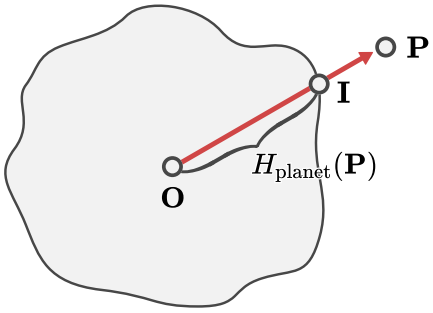


Figure 3.24: Planet heightmap.

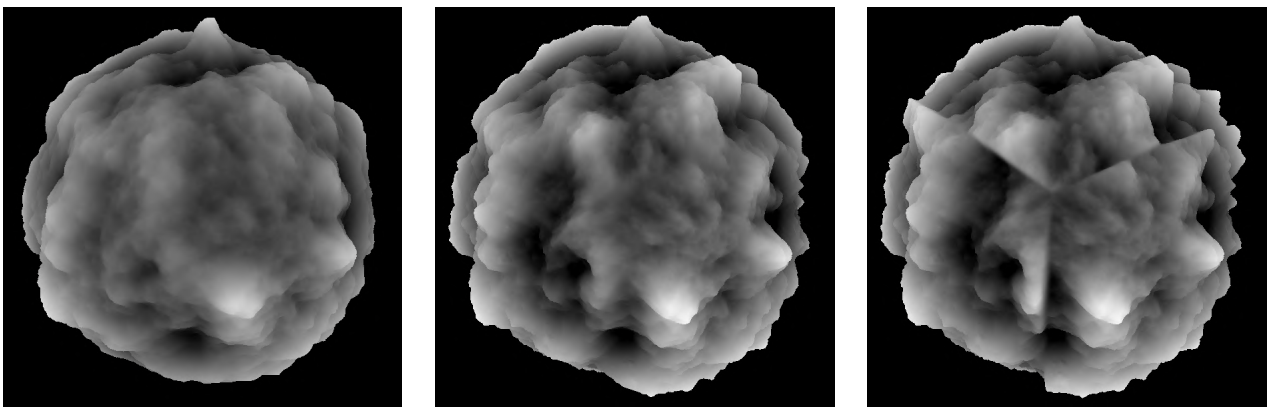
To calculate $H_{\text{planet}}(\mathbf{P})$, I used triplanar projection [33] to project 3 heightmaps onto a sphere, computed as:

$$\begin{aligned} H_{\text{planet}}(\mathbf{P}) = & H(\mathbf{P}.zy + f(\text{sgn}(\mathbf{P}.x))) \times w_x + \\ & H(\mathbf{P}.xz + f(\text{sgn}(\mathbf{P}.y))) \times w_y + \\ & H(\mathbf{P}.xy + f(\text{sgn}(\mathbf{P}.z))) \times w_z \end{aligned} \tag{3.25}$$

Here, H is the terrain heightmap (Section 3.2.1), the function f ensures that opposite sides of the planet do not share the same elevation, and the weights w_x, w_y, w_z are computed as follows:

$$w_x = |\hat{\mathbf{OP}}.x|^k, w_y = |\hat{\mathbf{OP}}.y|^k, w_z = |\hat{\mathbf{OP}}.z|^k \tag{3.26}$$

These weights are then normalized to sum up to 1. The parameter k controls the blending sharpness between heightmaps, as illustrated in Figure 3.25.



(a) $k = 1$ (b) $k = 10$ (c) $k = 100$

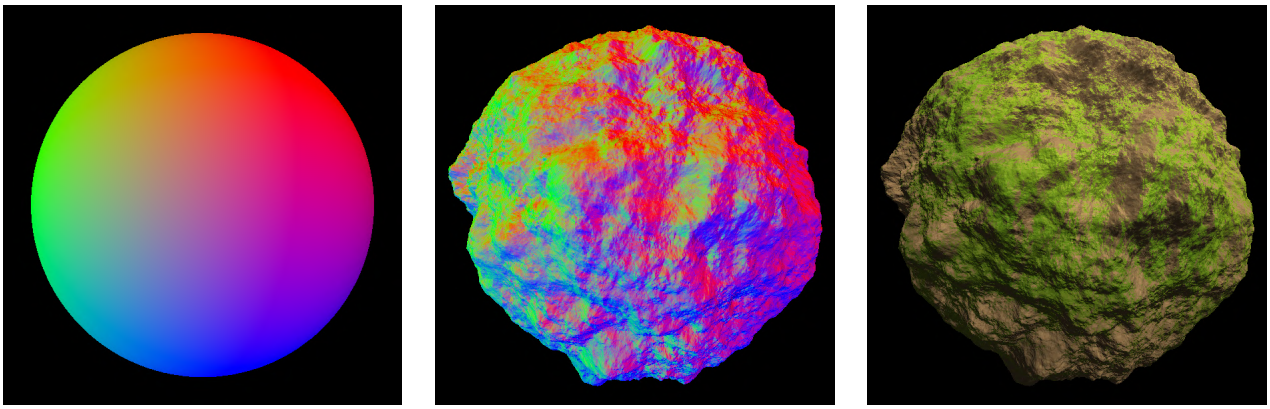
Figure 3.25: Comparison of triplanar-projected heightmaps with varying sharpness (k). Surface colors represent heightmap values, where darker colors indicate high values.

3.7.2 Ray Marching

Similar to terrain ray marching (Section 3.2.2), I used fixed-step ray marching for the planet's heightmap, incorporating early termination and dynamic step sizes. As an additional optimization, if the camera is outside the planet's bounding sphere, I begin ray marching from the camera ray's intersection with this sphere.

3.7.3 Procedural Coloring, Illumination and Shadows

The procedural coloring, illumination, and shadow techniques are largely similar to those used for flat terrain (Section 3.2), with the main difference being the normal calculations. Since the planet heightmap is a combination of three separate heightmaps, their normals need to be combined as well, a technique called triplanar normal mapping. First, the Whiteout Blend method [34] is used to blend the three normals along with the normal of the sphere at point $\mathbf{P}$, denoted as $\hat{\mathbf{OP}}$. The three normals are transformed from tangent space to world space and then blended using the weights for heightmap triplanar projection (Equation 3.26). The code for this normal blending process is provided in Appendix G. This approach ensures smooth and accurate normal calculations, as illustrated in Figure 3.26.



(a) Sphere normal. (b) Planet normal. (c) Shaded planet.

Figure 3.26: Visualizations of triplanar normal mapping.

3.7.4 Other Natural Elements

Water surfaces are modeled as spheres rather than planes. To render water, I first calculate the intersection between the camera ray and the water surface (Appendix B.4 details how to find the intersection between a ray and a sphere). Procedural coloring (Section 3.4.3) and illumination (Section 3.4.4) are then applied similarly to how they are for water on flat terrain.

The **atmospheric** density is still modeled as an exponential function based on altitude. However, the atmosphere is bounded by a sphere instead of a plane. To approximate the in-scattering equation, I only consider the segment of the ray inside the bounding sphere.

3.7.5 Example Output

Please refer to Appendix A.5 for example renders of procedural planets, and to `planet_navigation.mp4` for real-time navigation in a planet scene.

3.8 Input Controls

In this project, I implemented both a first-person, game-like camera control model [35] and trackball camera controls [36]. Additionally, I developed a callback system to overcome the limitations of GLFW. Implementation details are provided in Appendix E.

3.9 UI

I developed the application's UI using an object-oriented approach and a hierarchical structure, incorporating the Composite Design Pattern. I also implemented a save and load system for UI parameters and layout. Implementation details are available in Appendix F.

3.10 Repository Overview

```
/
├── Source/
│   ├── Input/
│   │   ├── Camera.cpp ..... Encapsulates camera transforms and parameters
│   │   ├── CameraController.cpp ..... Controls the camera
│   │   └── CallbackManager.cpp ..... Manages GLFW callbacks
│   ├── Pipeline/
│   │   ├── Main.cpp ..... Application entry point
│   │   ├── Window.cpp ..... Encapsulates GLFW windows
│   │   ├── Shader.cpp ... Handles OpenGL shaders, includes preprocessing and compilation
│   │   ├── FBO.cpp ..... Encapsulates OpenGL Framebuffer Objects
│   │   ├── VAO.cpp ..... Encapsulates OpenGL Vertex Array Objects
│   │   ├── VBO.cpp ..... Encapsulates OpenGL Vertex Buffer Objects
│   │   └── FPSCounter.cpp ..... Tracks FPS
│   ├── UI/
│   │   ├── App/ ..... Top-level UI class
│   │   ├── Panels/ ..... UI panels classes
│   │   └── Properties/ ..... UI properties classes
│   ├── Shaders/
│   │   ├── Main.vert/ ..... VS for the screen quad
│   │   ├── Main.frag/ ..... FS for the screen quad
│   │   ├── Debug.frag/ ..... Renders debug views
│   │   ├── Shading.frag/ ..... Handles lighting and shadows
│   │   ├── Raymarching.frag/ ..... Implements ray marching
│   │   ├── Clouds.frag/ ..... Clouds generation and volumetric rendering
│   │   ├── Trees.frag/ ..... Trees generation
│   │   ├── Planet.frag/ ..... Planet generation
│   │   ├── ValueNoise.frag/ ..... Implements value noise
│   │   ├── IntersectionDistanceError.frag/ ..... Calculates IDE incrementally
│   │   ├── HeightDifferenceError.frag/ ..... Calculates HDE
│   │   ├── fBm.frag/ ..... Implements of fBm
│   │   ├── Terrain.frag/ ..... Heightmap generation
│   │   ├── SphericalAtmosphere.frag/ ..... Atmosphere simulation for planets
│   │   ├── Atmosphere.frag/ ..... Atmosphere simulation
│   │   ├── Profiling.frag/ ..... Profiling macros
│   │   ├── TwoDSky.frag/ ..... 2D sky and clouds
│   │   ├── Encoding.frag/ ..... Encoding between floats and vec4s
│   │   ├── Water.frag/ ..... Water normal map
│   │   ├── Sun.frag/ ..... Calculates the sun direction and renders the sun disk
│   │   ├── SmoothStep.frag/ ..... Smoothstep functions
│   │   ├── Hash.frag/ ..... Hash functions
│   │   └── Motion.frag/ ..... Manages animations.
│   └── Evaluation/ ..... Python scripts and notebooks for evaluation
├── CMakeLists.txt ..... CMake script for compilation
└── LICENSE.txt ..... Project license
```

Figure 3.27: Repository overview. Header files (.h) are omitted for clarity. All code was written from scratch, except for the use of third-party libraries.

4 Evaluation

This chapter outlines the work done to evaluate the application, which is divided into three parts: **naturalness** (Section 4.1), **correctness** (Section 4.2), and **performance** (Section 4.3).

4.1 Naturalness

4.1.1 Terrain

Goal: Determine how different heightmap generation techniques affect the perceived naturalness of the rendered terrain scene.

Conditions Compared: The four heightmap generation models, listed in Table 4.1, are classified based on the incorporation of Dual Heightmap and Domain Distortion (Section 3.2.1).

Name	Dual Heightmap	Domain Distortion	Baseline
Dual Distorted	✓	✓	
Dual Original	✓	×	
Single Distorted	×	✓	
Single Original	×	×	✓

Table 4.1: Comparison of heightmap models

Method: To evaluate the perceived naturalness of each heightmap generation model, I conducted a pairwise comparison experiment with 14 participants. They viewed a video featuring 30 trials, each showing two 10-second side-by-side videos of the same terrain scene rendered in my application using different heightmap generation models. Participants then answered the prompt: "Considering only the geometry of the terrain (such as the overall form and finer details of the shape), which terrain in these images appears more natural?" This experiment considers five terrain scenes with various viewing conditions and environments. The final video was compiled using the `moviepy` Python library, with randomized pairwise comparisons.

Results: I used Matlab's `pwcmp` library [37], a set of functions for scaling pairwise comparisons, to calculate quality scores known as Just-Objectionable-Differences (JODs). Since JOD scores are relative, I chose the *Single Original* model, a single heightmap without domain distortion, as the baseline with a score of 0. A score of 1 JOD for another model indicates that 75% of participants found it more realistic than the baseline. Negative JOD scores suggest the baseline was preferred, while a score of 0 implies a lack of preference.

Figure 4.1 shows the average JOD scores for each heightmap generation method, aggregated over all scenes. The 95% confidence intervals, calculated through bootstrapping, are shown in the plot. The *Dual Distorted* model is perceived as the most natural, followed closely by *Dual Original*, with *Single Distorted* slightly outperforming the baseline.

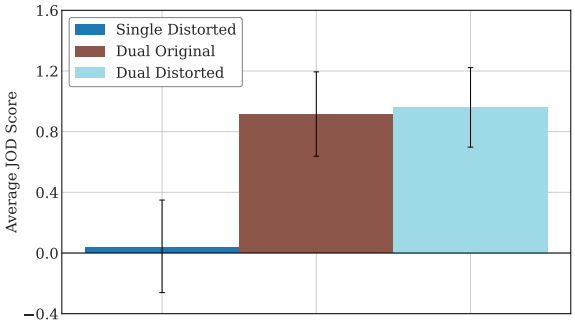


Figure 4.1: Average JOD scores for each heightmap generation model.

The scene-specific JOD scores in Figure 4.3 show that dual heightmap models consistently are perceived as more natural than single heightmap models across all scenes. However, models with domain distortion aren't always preferred over those without it.

For single heightmap models, distorted models are preferred, with the exception of Scene 5, where they are slightly less favored. In dual heightmap models, Scenes 1 and 4 stand out because domain-distorted models are perceived as less natural. Observing the scenes in Figure 4.2, this may be due to domain distortion causing some areas to appear overly stretched. In general, models with domain distortion are perceived similarly to those without. This is because I had to keep the distortion strength low to avoid highly unnatural results. Furthermore, its effects vary across different regions, making them less noticeable.

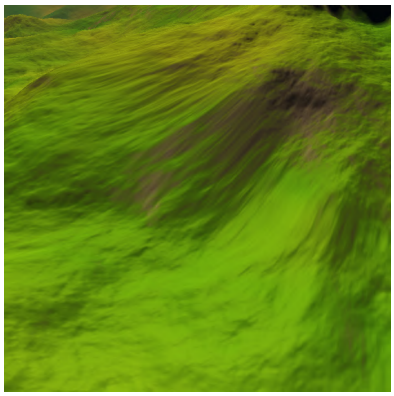


Figure 4.2: Stretched areas.

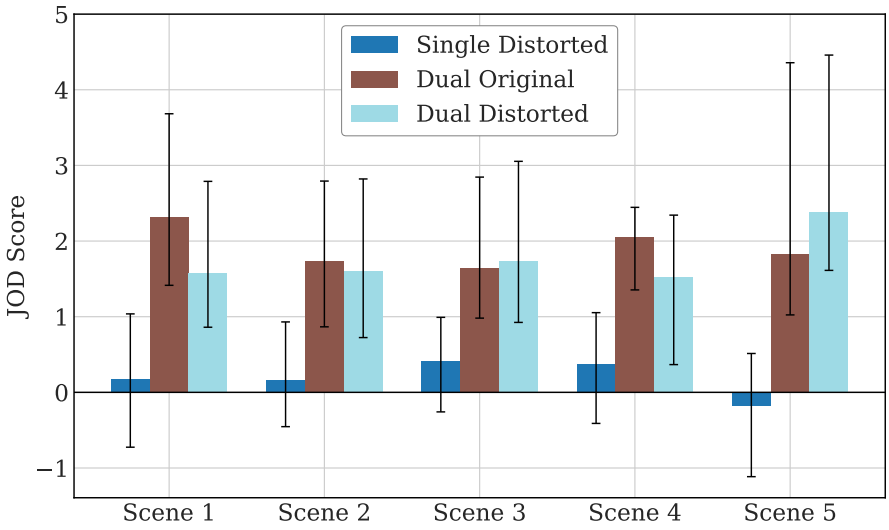


Figure 4.3: JOD scores broken down for each scene presented.

I used `pwcmp` to test for two-tailed statistical significance in the differences between JOD scores, shown in Figure 4.4. The results indicate a significant difference between dual and single heightmap models across all scenes. However, there is no significant difference between models with and without domain distortion.

Overall, the dual-heightmap technique significantly improves perceived naturalness, making it preferable to the single-heightmap approach. Although domain distortion does not significantly improve perceived naturalness, it can still be helpful for creating intriguing, alien landscapes, as shown in Figure 3.7.

4.1.2 Atmosphere

Goal: To compare the perceived naturalness of the simulated atmosphere (Section 3.5) with the basic implementation of the atmosphere (Appendix D.1).

Method: I conducted a pairwise comparison experiment similar to the one described in Section 4.1.1, comparing renders from the two atmospheric implementations. The experiment used 10 scenes with varying sun directions to represent different times of day and viewing conditions. Participants were asked, "Which render provides a more natural overall atmosphere, including sky color and depiction of distant terrain?"

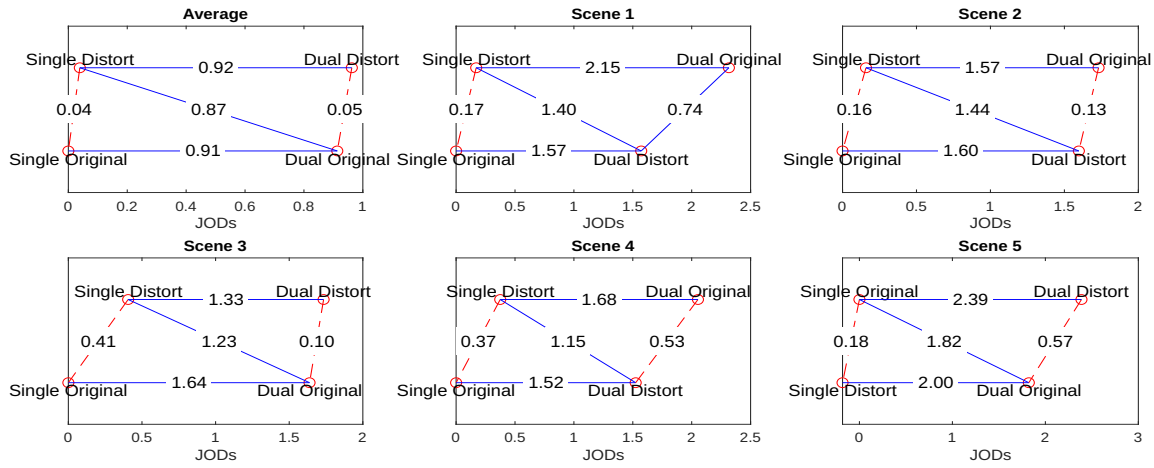


Figure 4.4: Visualization of JOD score differences: Continuous lines indicate statistically significant differences between models, while dashed lines signify no significant difference.

Results: Using Python for data analysis, Figure 4.5 shows the probability p of participants favoring the simulated atmosphere over the basic implementation as more natural, including 95% confidence intervals for each scene and an average result. The probabilities generally exceed 0.5, with an average $p = 0.86$.

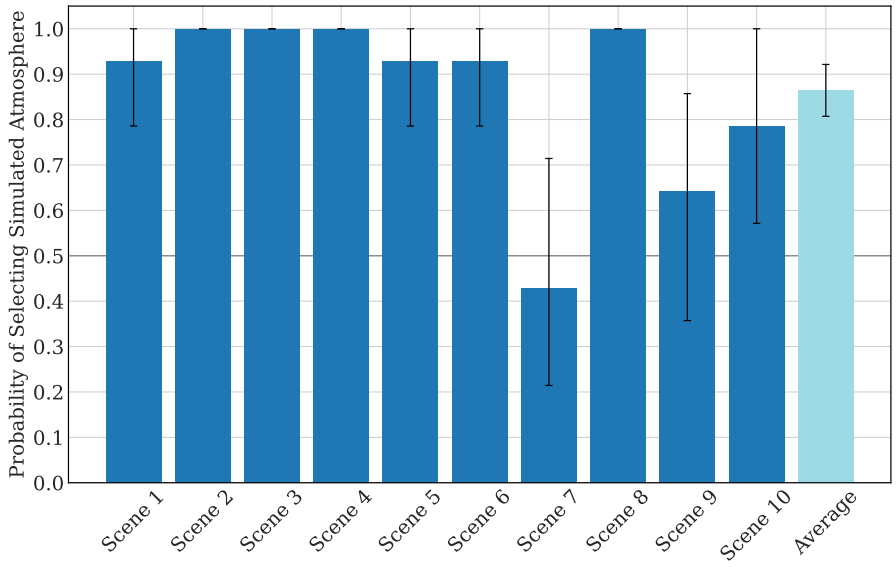
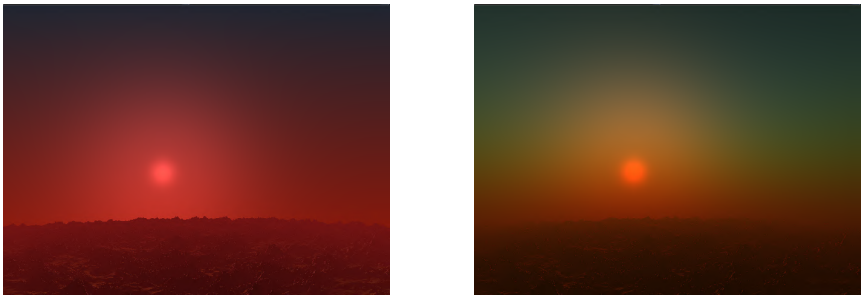


Figure 4.5: Probability of choosing the simulated atmosphere as more natural than the basic implementation, broken down per scene.

I performed a one-tailed Binomial statistical test at a 2.5% significance level to test H_0: the simulated atmosphere is not perceived as more natural than the basic implementation ($p \leq 0.5$). All scenes except Scene 7 and 9 can reject H_0 with 97.5% confidence, implying that the simulated atmosphere is perceived as more natural. Scene 7 stands out as more people perceived the basic implementation as more natural. Observing Scene 7 in Figure 4.6, this is likely due to the red hue being associated with a sunset.

Overall, the evidence supports the rejection of H_0 for most scenes and on average, suggesting the simulated atmosphere is perceived as more natural and preferable to the basic implementation.



(a) Basic implementation (b) Simulated atmosphere

Figure 4.6: Renders from Scene 7, for both atmosphere implementations

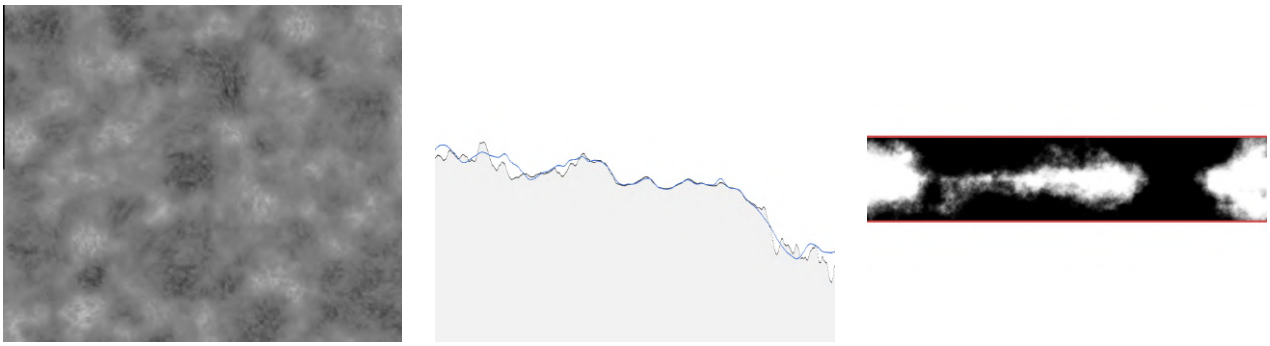
4.2 Correctness

4.2.1 Visual Debugging

Goal: Ensure correct implementation of procedural generation and rendering of natural elements in this project.

Methods and results: To validate the procedural generation and rendering methods, I utilized visual debugging across various components. For the terrain heightmap, I created debug views that display the heightmap in a two-dimensional space (Figure 4.7a) and display cross-sectional slices along the x-axis in one dimension (Figure 4.7b). This approach facilitates verification of the correctness of terrain generation methods discussed in Section 3.2.1, independent of ray marching or illumination models. Similarly, for debugging the procedural generation of clouds, I created a view that displays a slice of the density map along the x-axis (Figure 4.7c).

To verify the correctness of terrain ray marching independent of illumination, I created a debug view that displays the depth map (Figure 4.8a). Similarly, to validate atmosphere simulation, I developed a debug view that shows the optical depth (Section 3.5.2) of the view rays (Figure 4.8b). To check the procedural generation of planets, I created a view that displays a slice of the planet's heightmap in 2D (Figure 4.8c). Moreover, for testing normal calculations, I employed a simple Lambertian diffuse component, which is visualized along the outline. Furthermore, I visualized the atmospheric density using a color gradient, and tested ray intersections with the sphere by rendering circle and line gizmos.

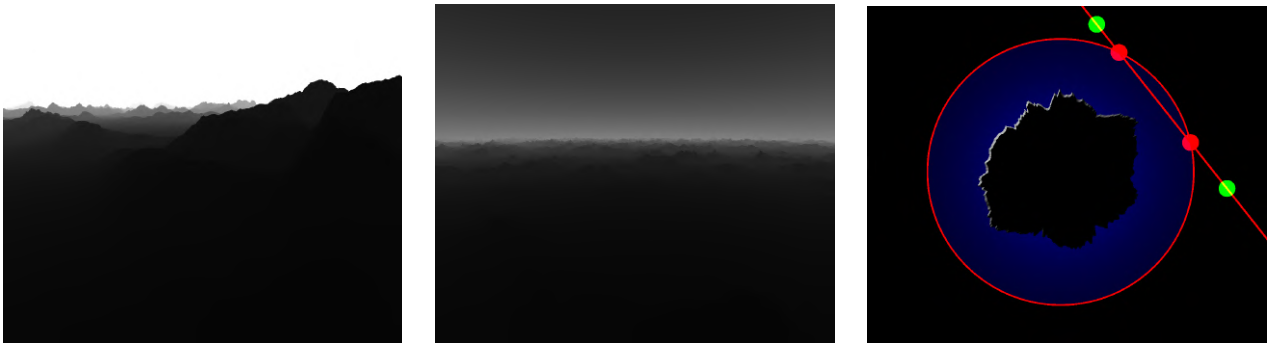


(a) Debug view of the heightmap in 2D. Brighter colors correspond to higher elevations.

(b) Slice of the heightmap along the x-axis. The blue curve is the global heightmap in the dual heightmap approach.

(c) Debug view of the cloud density map along the x-axis. Red lines denote the boundaries of the cloud bounding box.

Figure 4.7: Debug views.



(a) Visualizing the depth map from ray marching. Brighter colors correspond to longer distances from the camera to surface intersections.

(b) Visualizing the optical depth of the camera rays. Brighter colors correspond to higher optical depth values.

(c) Visualizations to test various components related to procedural planets generation.

Figure 4.8: Debug views.

4.2.2 Terrain Ray Marching Correctness

Error Metrics for Terrain Ray Marching

To assess the correctness of terrain ray marching, I define two types of errors:

1. **Height Difference Error (HDE)**. This measures the vertical distance between the intersection point determined by ray marching and the corresponding point on the heightmap, as shown in Figure 4.9. The HDE is defined for a point $\mathbf{P}$ as:

$$\text{HDE}(\mathbf{P}) = |\mathbf{P}.y - H(\mathbf{P}).y| \tag{4.1}$$

2. **Intersection Distance Error (IDE)**. This measures the distance between the intersection found by ray marching and the actual ground truth intersection, as illustrated in Figure 4.9. A notable special case includes **Missed Intersections**, which occur when ray marching fails to detect an intersection that actually exists.

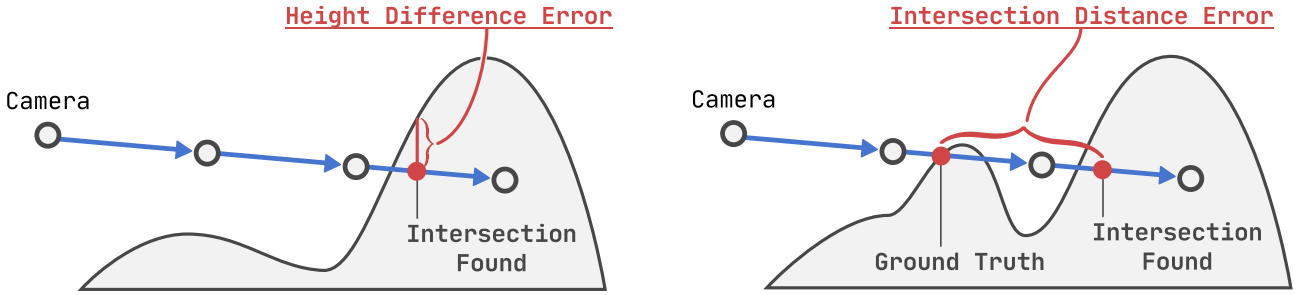


Figure 4.9: *Left*: illustrates the HDE, *Right*: illustrate the IDE.

Ground Truth Determination Because it is impractical to analytically determine intersections between a ray and a heightmap, I approximate the ground truth intersection using fixed-step ray marching with a small step size and a high number of steps. Due to its computational intensity, this process uses incremental rendering over several frames. Each frame encodes the progress of ray marching into the framebuffer using bit operations on a 32-bit `vec4` to encode and decode float values.

Interpretation of Errors HDE reflects the precision of ray marching: a high HDE suggests a significant vertical misalignment from the terrain surface, potentially causing flickering artifacts of the terrain surface as the camera moves or rotates, as demonstrated in `raymarch_artifact.mp4`. **IDE** reflects the accuracy of the intersection calculations: a high IDE

indicates that some terrain features might have been erroneously skipped, leading to incorrect representations of terrain shape, with parts potentially missing. **Missed Intersections** represent a critical error where the ray marching algorithm fails to identify an intersection, even though one exists. This issue leads to prominent gaps in the rendered scene, where the sky is visible instead of the expected terrain.

Experiments

Goal: Evaluate the correctness of terrain ray marching within this project by examining how variations in ray marching parameters influence the results.

Parameters Considered: Four key ray marching parameters are considered: a—initial step size, b—scaling factor relative to distance, c—scaling factor relative to height above terrain, and k—number of steps in the binary search. As described in Section 3.2.2, the step size s_i for each iteration i of the terrain ray marching process is calculated as follows:

$$s_i = a + b \cdot d_i + c \cdot (\mathbf{P}_i.y - H(\mathbf{P}_i.xz)) \tag{4.2}$$

Once a point goes below the heightmap, k binary search steps are performed to determine the final intersection point.

Method and Results: I carried out four experiments using my laptop with an AMD Ryzen 7 4800H CPU, 16GB of RAM, and an NVIDIA GeForce RTX 2060 GPU. In each experiment, I measured the IDE and HDE while varying one ray marching parameter at a time, keeping all other parameters constant. Additionally, I measured the frame time averaged over 100 consecutive frames, with the terrain being the only natural element active in the scene. The resolution was maintained at 600×600 pixels throughout the experiments.

In **Experiment 1**, the initial step size a was increased for $a \in \{0.1, 1, 5, 10, 20, 50, 100\}$, while other parameters were held constant: $b = 0.004$, $c = 50$, and $k = 0$. As shown in the plot in Figure 4.10, the average HDE, IDE, and the number of missed intersections all increase linearly with a. This is due to larger a leading to larger step sizes, thereby increasing the likelihood of missing intersections and reducing precision when intersections are detected. Additionally, the average frame time decreases exponentially as a increases. This reduction is due to the larger step sizes allowing ray marching to either find intersections with fewer steps or to quickly determine the absence of intersections.

In **Experiment 2**, the scaling factor relative to distance, b, was increased for $b \in \{1, 2, 4, 10, 20, 50, 100\} \times 10^{-3}$, while other parameters were held constant: $a = 5$, $c = 50$, $k = 0$. As shown in Figure 4.11, similar to Experiment 1, the average HDE, IDE, and the number of missed intersections all linearly increase with b. Additionally, the average frame time decreases exponentially as b increases. These trends are attributed to the increase in step sizes caused by higher b values.

Moreover, I examined the impacts of the scaling factor b on intersections at various distances from the camera. I classified intersections into three categories based on their proximity: Short Range, Medium Range, and Long Range. According to Figure 4.12, the increase in the average HDE, IDE, and the number of missed intersections is significantly slower for Short Range intersections. This slower increase can be attributed to the smaller step size increments that result from b when the intersection point is closer to the camera.

In **Experiment 3**, the scaling factor relative to height above ground, c, was increased for $c \in \{0, 1, 2, 4, 10, 20, 50, 100\}$, while other parameters were held constant: $a = 5$, $b = 0.005$, $k = 0$. As shown in Figure 4.13, the average HDE, IDE, and number of missed intersections have minimal variation as c increases. This occurs because c significantly influences the step size only when the point is substantially elevated above the terrain, which typically suggests a

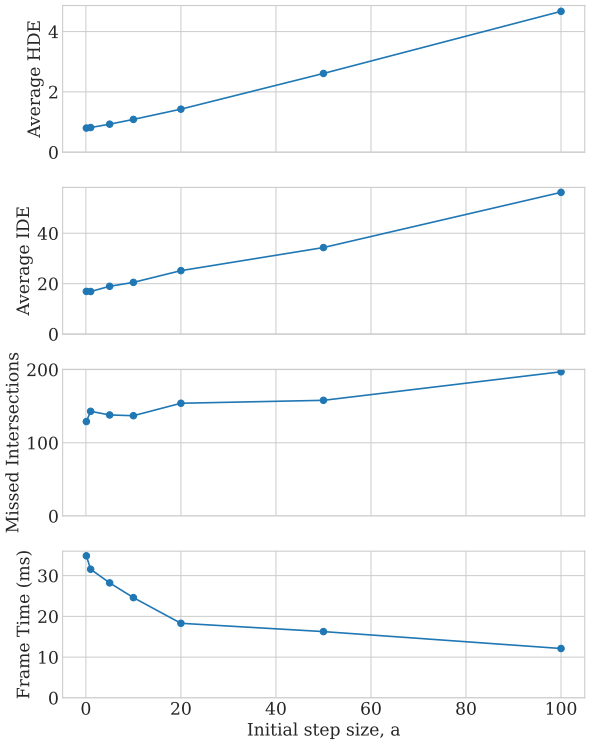


Figure 4.10: Results of Experiment 1.

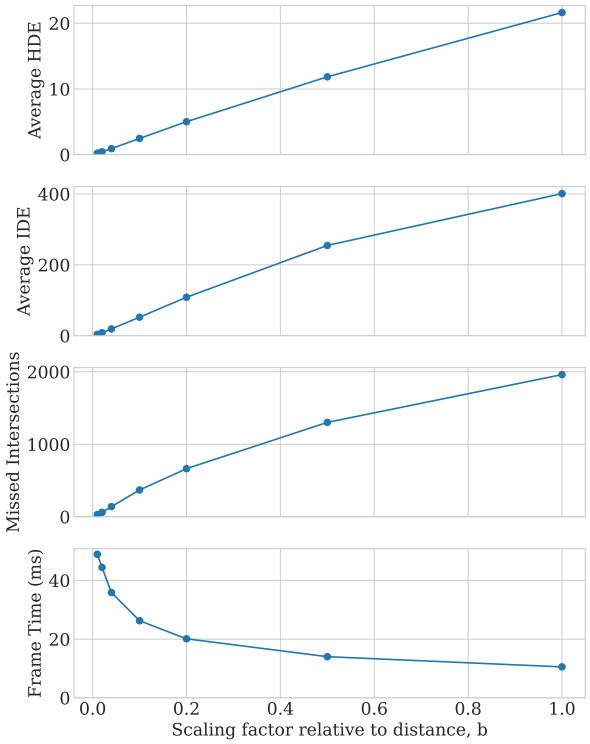


Figure 4.11: Results of Experiment 2.

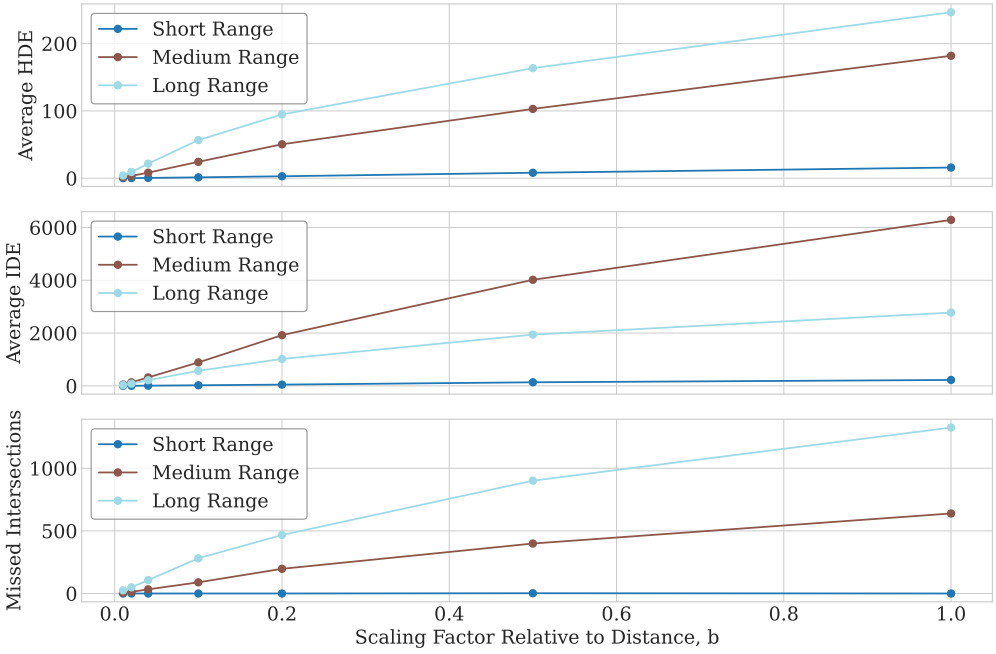


Figure 4.12: Results of Experiment 2, for intersections with varying distances from the camera.

lower likelihood of encountering terrain along the ray path. Additionally, average frame time decreases exponentially as a increases. This is because stepsizes are increased when points are high above the terrain surface, allowing ray marching to either find intersections with fewer steps or to quickly determine the absence of intersections.

In **Experiment 4**, the number of binary search steps, k, was increased for $k \in \{0, 1, 2, 4, 6, 10, 15, 20\}$, while other parameters were held constant: $a = 5$, $b = 0.01$ $c = 50$. As observed in Figure 4.14, increasing k leads to an exponential decrease in the average HDE, which indicates a significant improvement in ray marching precision with a small number of

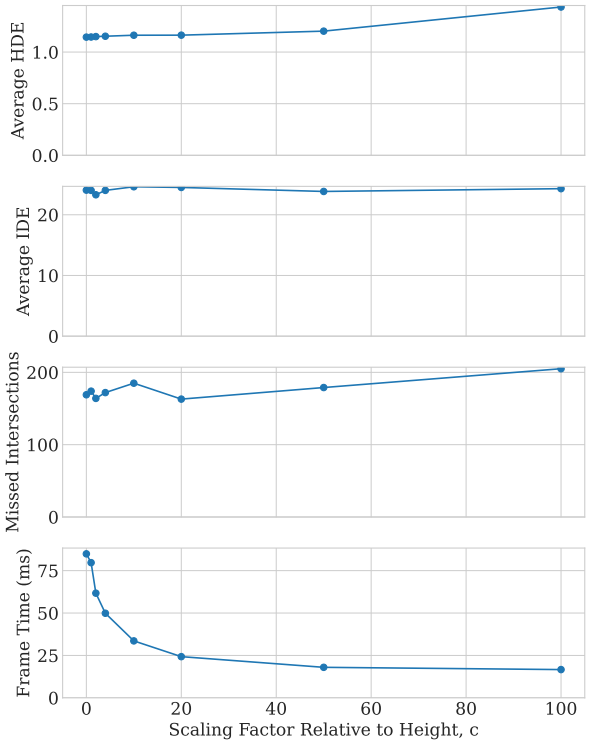


Figure 4.13: Results of Experiment 3.

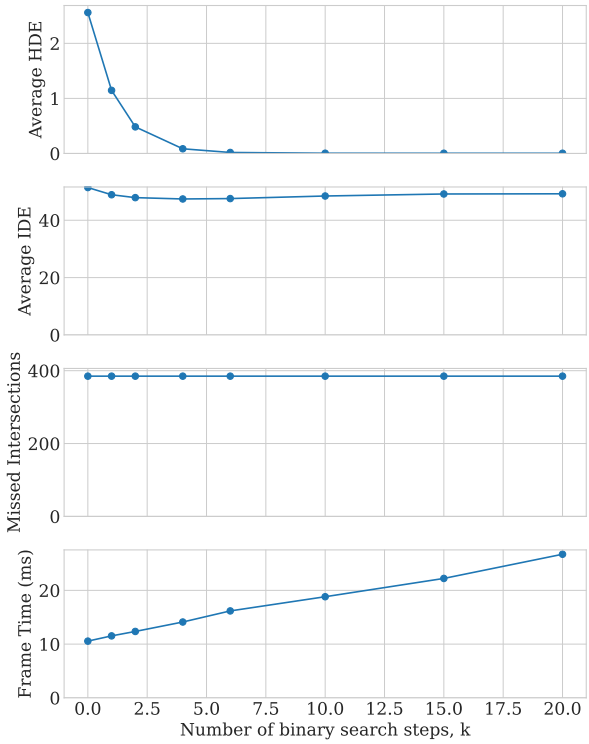


Figure 4.14: Results of Experiment 4.

steps, but with diminishing returns. This is due to each binary search step halving the search range for the intersection.

However, it is also observed that increases in k only affect the average IDE within a narrow range (approximately 50 ± 3), and do not alter the number of missed intersections. This is because the binary search is conducted only after an initial intersection is detected, limiting its impact to refining the position between the last and the second-to-last points. Additionally, the average frame time increases linearly with k. This is due to the greater number of binary search steps increases the number of heightmap evaluations.

4.3 Performance

4.3.1 Visual Profiling

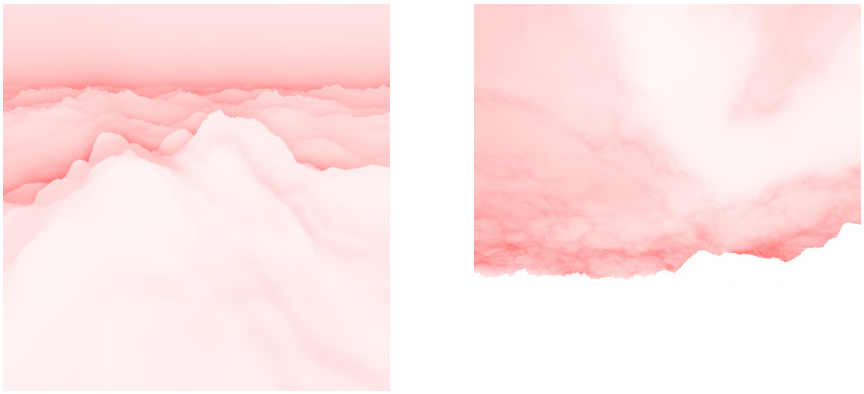
Goal: Understand the computational load distribution across different regions to render various natural elements.

Methods and Results: I developed heatmap-like debug views to analyze the computational load distribution across different regions. Specifically, I created views for the number of ray marching steps required for terrain and clouds. These heatmaps, illustrated in Figure 4.15, effectively highlight areas where rendering demands are highest for terrain and clouds.

4.3.2 Resolution and Natural Elements

Goal: Understand how varying resolutions and different natural elements affect the performance of the application, relative to the timing constraints typical in real-time graphics applications.

Method: I conducted an experiment on my laptop with an AMD Ryzen 7 4800H CPU, 16GB of RAM, and an NVIDIA GeForce RTX 2060 GPU. I measured the average frame time—the time required to render a single frame, averaged over 100 consecutive



(a) Terrain ray marching steps. (b) Cloud ray marching steps.

Figure 4.15: Heatmap debug views. White regions represent areas requiring fewer steps, whereas red regions represent areas requiring more steps.

frames—at increasing resolutions. Resolutions tested ranged from $n \times n$ pixels, where $n \in \{50, 100, 200, 300, 400, 600, 800, 1000, 1500\}$. For each resolution, I recorded the frame time for individual natural elements: terrain, atmosphere, clouds, and trees.

Results: The results, illustrated in Figure 4.16, demonstrate that frame time scales quadratically with n and linearly with resolution ($n \times n$). This pattern is attributable to the pixel-dependent processes involved in ray marching, coloring, and illumination. A separate plot in Figure 4.17 details the frame times for each natural element. The data indicates that trees require the most time to render, primarily due to the computational demands of evaluating their SDF and the necessity of computing the distances to adjacent elements due to domain repetition. Clouds also show significant rendering times, followed by terrain. The atmosphere was the least taxing on frame time, benefiting from simpler exponential density functions and fewer required steps to achieve accurate approximations using Riemann sums.

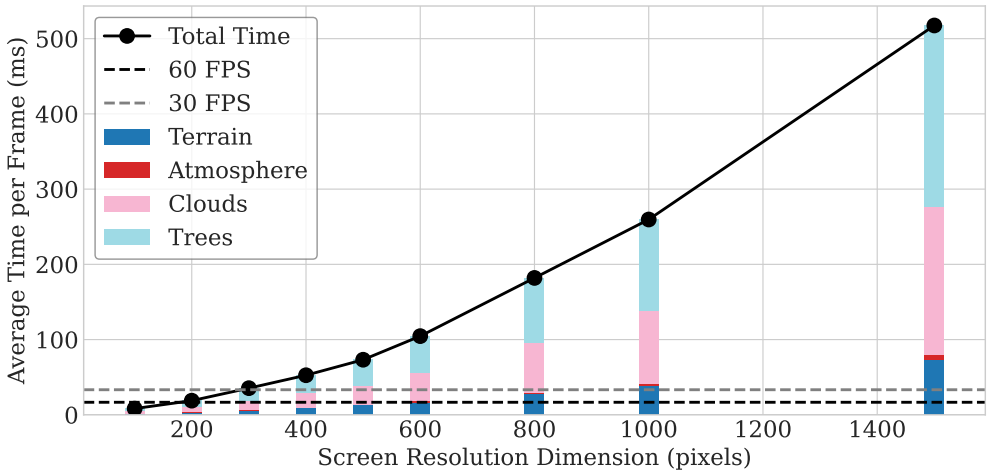


Figure 4.16: Plot of average frame time versus resolution dimension, with bars indicating the contribution of different natural elements to the frame time.

In real-time graphics applications, achieving a steady 60 Frames-Per-Second (FPS) is crucial for smooth user experiences on modern hardware [38]. Human perception needs at least 10 to 12 FPS to perceive motion [39], and film standards regard 24 to 30 FPS as adequate for fluid motion without noticeable stutters [40].

In my application, rendering all natural elements results in high frame rates only at very low resolutions, with performance dropping below 60 FPS at resolutions around $n \approx 200$ and falling under 30 FPS at $n \approx 300$. Excluding trees improves performance, with the frame rate dropping

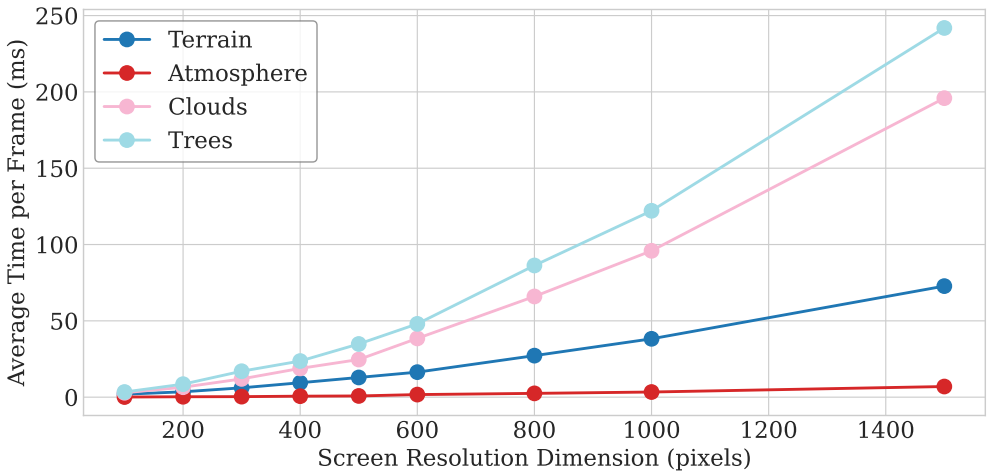


Figure 4.17: Plot of average frame time of different natural elements versus resolution dimension.

below 30 FPS when $n \approx 500$. Such low resolutions may be suitable for preview purposes in offline rendering but are inadequate for real-time applications like games. However, rendering only terrain and atmosphere maintains frame rates above 30 FPS up to a more acceptable $n \approx 900$, which is more reasonable for real-time applications.

4.3.3 Memory Usage

I used RenderDoc, a graphics debugging tool, to assess the **static memory allocation**—memory allocated for resources that remain constant during runtime. This method does not account for dynamic memory allocation, as my application does not generate resources such as textures dynamically.

The results shows that the application allocates 4 textures and 4 buffers, totaling 9.15 MB in GPU memory for both. This allocation is relatively modest for real-time applications such as games. The reason for the low memory consumption is that my application primarily uses implicit representations, and the only mesh is a screen quad, which requires minimal memory for vertex and index buffers. Additionally, since coloring and texturing are handled procedurally, the only significant texture usage comes from the UI.

4.3.4 Scene Storage

I analyzed the scene storage by measuring the number of parameters (floats, ints, or bools) in the JSON save file, totaling 301 parameters with a file size of approximately 27.0 KB. This file size is relatively small when compared to the sizes typically associated with mesh data or Neural Implicit Representations. However, the use of JSON for scene storage has limited compatibility as it can only be loaded and interpreted by my application.

To enhance compatibility, I developed an alternative method that generates a fragment shader file, substituting most uniforms with values in the JSON save file. This approach is more compatible since the shader can be executed on any system that supports GLSL and is configured with a screen quad. The resulting shader file contains 2,339 non-empty lines and is approximately 82.9 KB in size. However, this method does not support interactive adjustments of parameter values at runtime, as the uniforms are converted into constants.

5 Conclusions

This project was a success: I met all criteria outlined in Section 2.4.1, and developed most of the extension deliverables listed in Table 2.1. The project's objective was to create an application capable of rendering procedurally generated natural environments in real time using ray marching of implicit representations. My application succesfully achieves them, enabling users to specify parameters that can be saved and loaded, allowing them to generate, view, and navigate real-time environments that feature terrain, trees, water, volumetric clouds, and a simulated atmosphere. The renderer uses ray marching to find intersections with implicit scene representations accurately and efficiently. Additionally, the application has been expanded to apply procedural generation and rendering techniques to procedural planets.

Comprehensive evaluation validated the correctness of terrain ray marching and the visual correctness of other components. I have also evaluated the perceived naturalness of the terrain and atmosphere and the application's performance. The dual heightmap and simulated atmosphere significantly improved perceived naturalness. Optimizations in ray marching enabled the application to render natural environments up to a limited resolution under real-time constraints, while requiring minimal memory and storage.

5.1 Key Lessons

This has been the largest and most impressive software engineering project I've undertaken independently so far. I've learned the importance of maintaining a modular and standardized codebase to ensure maintainability, even as a solo developer. I also discovered that visual debugging is highly effective for testing individual components, given the limited debugging and testing tools available for shader programming. Overall, this project has been an enjoyable and enriching experience, especially as I watched my application gradually become more fully-fledged, as shown in Figure 5.1.

(a) Early stages (b) Core finished (c) Final state

Figure 5.1: Progress of the project.

5.2 Directions for Future Work

I propose several areas for future exploration to enhance the naturalness of procedural environments, which were beyond this project's scope due to time and space constraints:

- Incorporate complex terrain features such as caves and overhangs by generating them through shape grammar [41], or by using a 3D representation instead of a heightmap [42].
- Improve the water illumination model by adding reflections, achieved by continuing ray marching from the point of intersection.
- Apply clouds and trees to procedural planets.

Bibliography

[1] Eric Galin et al. "A Review of Digital Terrain Modeling". en. In: *Computer Graphics Forum* 38.2 (2019). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13657, pp. 553–577. ISSN: 1467-8659. DOI: 10.1111/cgf.13657. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13657 (visited on 05/10/2024).

[2] Benoit B. Mandelbrot and John W. Van Ness. "Fractional Brownian Motions, Fractional Noises and Applications". en. In: *SIAM Review* 10.4 (Oct. 1968), pp. 422–437. ISSN: 0036-1445, 1095-7200. DOI: 10.1137/1010093. URL: http://epubs.siam.org/doi/10.1137/1010093 (visited on 05/09/2024).

[3] Benoit B. Mandelbrot and John A. Wheeler. "The Fractal Geometry of Nature". In: *American Journal of Physics* 51.3 (Mar. 1983), pp. 286–287. ISSN: 0002-9505. DOI: 10.1119/1.13295. URL: https://doi.org/10.1119/1.13295 (visited on 05/10/2024).

[4] Ken Perlin. "An image synthesizer". In: *ACM SIGGRAPH Computer Graphics* 19.3 (July 1985), pp. 287–296. ISSN: 0097-8930. DOI: 10.1145/325165.325247. URL: https://dl.acm.org/doi/10.1145/325165.325247 (visited on 05/10/2024).

[5] Jacob Olsen. "Realtime Procedural Terrain Generation". In: (Jan. 2004).

[6] F. K. Musgrave, C. E. Kolb, and R. S. Mace. "The synthesis and rendering of eroded fractal terrains". In: *ACM SIGGRAPH Computer Graphics* 23.3 (July 1989), pp. 41–50. ISSN: 0097-8930. DOI: 10.1145/74334.74337. URL: https://dl.acm.org/doi/10.1145/74334.74337 (visited on 05/10/2024).

[7] Ryan John Spick and James Alfred Walker. *Realistic and Textured Terrain Generation using GANs*. en. Proceedings Paper. Publisher: ACM. Oct. 2019. URL: https://eprints.whiterose.ac.uk/153088/ (visited on 05/10/2024).

[8] William E. Lorensen and Harvey E. Cline. "Marching cubes: A high resolution 3D surface construction algorithm". In: *ACM SIGGRAPH Computer Graphics* 21.4 (Aug. 1987), pp. 163–169. ISSN: 0097-8930. DOI: 10.1145/37402.37422. URL: https://dl.acm.org/doi/10.1145/37402.37422 (visited on 04/29/2024).

[9] Inigo Quilez. *Raymarching Terrain*. en. URL: https://iquilezles.org/articles/terrainmarching/ (visited on 04/29/2024).

[10] Simon Blom. *A Comparison Between Rendering Techniques For Billboards In DirectX 11*. eng. 2021. URL: https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-302261 (visited on 05/10/2024).

[11] Ben Carey. "Procedural Forest Generation with L-System Instancing". en. In: ().

[12] Stanley Osher and Ronald Fedkiw. "Signed Distance Functions". en. In: *Level Set Methods and Dynamic Implicit Surfaces*. Google-Books-ID: i4bfBwAAQBAJ. Springer Science & Business Media, Apr. 2006, pp. 17–22. ISBN: 978-0-387-22746-7.

[13] Jasper Flick. *Waves*. en. URL: https://catlikecoding.com/unity/tutorials/flow/waves/ (visited on 05/10/2024).

[14] Jasper St. Pierre. *Deconstructing the water effect in Super Mario Sunshine — Clean Rinse*. en-US. Mar. 2018. URL: https://blog.mecheye.net/2018/03/deconstructing-the-water-effect-in-super-mario-sunshine/ (visited on 05/10/2024).

[15] Thomas Poulet. *The Witness: frame analysis part 1*. en-us. URL: http://blog.thomaspoulet.fr/the-witness-frame-part-1/ (visited on 05/10/2024).

[16] *Skybox Basics - Valve Developer Community*. URL: https://developer.valvesoftware.com/wiki/Skybox_Basics (visited on 05/10/2024).

[17] Fredrik Haggstrom. "Real-time rendering of volumetric clouds". en. In: (2018).

[18] Tomoyuki Nishita et al. "Display of the earth taking into account atmospheric scattering". In: *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. SIGGRAPH '93. New York, NY, USA: Association for Computing Machinery, Sept. 1993, pp. 175–182. ISBN: 978-0-89791-601-1. DOI: 10.1145/166117.166140. URL: https://dl.acm.org/doi/10.1145/166117.166140 (visited on 05/09/2024).

[19] Ton Dieker. "Simulation of fractional Brownian motion". en. In: (2004).

[20] Patricio Gonzalez Vivo and Jen Lowe. *Fractal Brownian Motion*. URL: https://thebookofshaders.com/13/ (visited on 04/29/2024).

[21] Spatial Team. *The Main Benefits and Disadvantages of Polygonal Modeling*. en-us. Dec. 2019. URL: https://blog.spatial.com/the-main-benefits-and-disadvantages-of-polygonal-modeling (visited on 05/09/2024).

[22] Stanley Osher and Ronald Fedkiw. "Implicit Functions". en. In: *Level Set Methods and Dynamic Implicit Surfaces*. Google-Books-ID: i4bfBwAAQBAJ. Springer Science & Business Media, Apr. 2006, pp. 3–16. ISBN: 978-0-387-22746-7.

[23] John Hart. "Sphere Tracing: A Geometric Method for the Antialiased Ray Tracing of Implicit Surfaces". In: *The Visual Computer* 12 (June 1995). DOI: 10.1007/s003710050084.

[24] Bui Tuong Phong. "Illumination for Computer Generated Pictures". en. In: 18.6 (1975).

[25] Dorian Iten. *Understanding the Fresnel Effect*. en-US. Dec. 2018. URL: https://www.dorian-iten.com/fresnel/ (visited on 05/04/2024).

[26] Ian Dunn and Zoë Wood. *Schlick's Approximation*. URL: https://graphicscompendium.com/raytracing/11-fresnel-beer (visited on 05/04/2024).

[27] Dai Clegg and Richard Barker. *Case Method Fast-Track: A Rad Approach*. USA: Addison-Wesley Longman Publishing Co., Inc., May 1994. ISBN: 978-0-201-62432-8.

[28] Inigo Quilez. *Ellipsoid SDF*. en. URL: https://iquilezles.org/articles/ellipsoids/ (visited on 04/29/2024).

[29] Inigo Quilez. *Domain Repetition*. en. URL: https://iquilezles.org/articles/sdfrepetition/ (visited on 04/29/2024).

[30] James F. Blinn. "Simulation of wrinkled surfaces". In: *ACM SIGGRAPH Computer Graphics* 12.3 (Aug. 1978), pp. 286–292. ISSN: 0097-8930. DOI: 10.1145/965139.507101. URL: https://dl.acm.org/doi/10.1145/965139.507101 (visited on 05/09/2024).

[31] Andrew T. Young. "Rayleigh scattering". EN. In: *Applied Optics* 20.4 (Feb. 1981). Publisher: Optica Publishing Group, pp. 533–535. ISSN: 2155-3165. DOI: 10.1364/AO.20.000533. URL: https://opg.optica.org/ao/abstract.cfm?uri=ao-20-4-533 (visited on 05/09/2024).

[32] Deborah Hughes–Hallett et al. *Applied Calculus*. English. 4th Edition. Hoboken, NJ: John Wiley & Sons, Dec. 2009. ISBN: 978-0-470-17052-6.

[33] Jasper Flick. *Triplanar Mapping*. en. URL: https://catlikecoding.com/unity/tutorials/advanced-rendering/triplanar-mapping/ (visited on 05/01/2024).

[34] Christopher Oat. "Animated wrinkle maps". In: *ACM SIGGRAPH 2007 courses*. SIGGRAPH '07. New York, NY, USA: Association for Computing Machinery, Aug. 2007, pp. 33–37. ISBN: 978-1-4503-1823-5. DOI: 10.1145/1281500.1281667. URL: https://dl.acm.org/doi/10.1145/1281500.1281667 (visited on 05/04/2024).

[35] *First-person (video games)*. en. Page Version ID: 1218717225. Apr. 2024. URL: https://en.wikipedia.org/w/index.php?title=First-person_(video_games)&oldid=1218717225 (visited on 05/09/2024).

[36] *Object Mouse Trackball - OpenGL Wiki*. URL: https://www.khronos.org/opengl/wiki/Object_Mouse_Trackball (visited on 05/01/2024).

[37] Maria Perez-Ortiz and Rafal K. Mantiuk. *A practical guide and software for analysing pairwise comparison experiments*. arXiv:1712.03686 [cs, stat]. Dec. 2017. DOI: 10.48550/arXiv.1712.03686. URL: http://arxiv.org/abs/1712.03686 (visited on 05/01/2024).

[38] Amin Banitalebi Dehkordi, Mahsa Pourazad, and Panos Nasiopoulos. "The Effect of Frame Rate on 3D Video Quality and Bitrate". In: *3D Research* 6 (Feb. 2015). DOI: 10.1007/s13319-014-0034-3.

[39] Paul Read and Mark-Paul Meyer. "Chapter 4.7.1: Frame rates". en. In: *Restoration of Motion Picture Film*. Google-Books-ID: jzbUUL0xJAEC. Elsevier, Aug. 2000, pp. 24–26. ISBN: 978-0-08-051619-6.

[40] Levi Tijerina. *What is Frame Rate, and why does it matter? (24fps vs. 30fps)*. en-US. Feb. 2021. URL: https://gamut.io/why-frame-rate-matters-24fps-vs-30fps/ (visited on 05/12/2024).

[41] Axel Paris et al. "Terrain Amplification with Implicit 3D Features". In: *ACM Transactions on Graphics* 38.5 (Sept. 2019), 147:1–147:15. ISSN: 0730-0301. DOI: 10.1145/3342765. URL: https://dl.acm.org/doi/10.1145/3342765 (visited on 04/29/2024).

[42] Ryan Geiss. *Chapter 1. Generating Complex Procedural Terrains Using the GPU*. en-US. URL: https://developer.nvidia.com/gpugems/gpugems3/part-i-geometry/chapter-1-generating-complex-procedural-terrains-using-gpu (visited on 04/29/2024).

[43] Inigo Quilez. *Normals for an SDF*. en. URL: https://iquilezles.org/articles/normalsSDF/ (visited on 04/29/2024).

A Example Renders

A.1 Example Renders from the Core Version

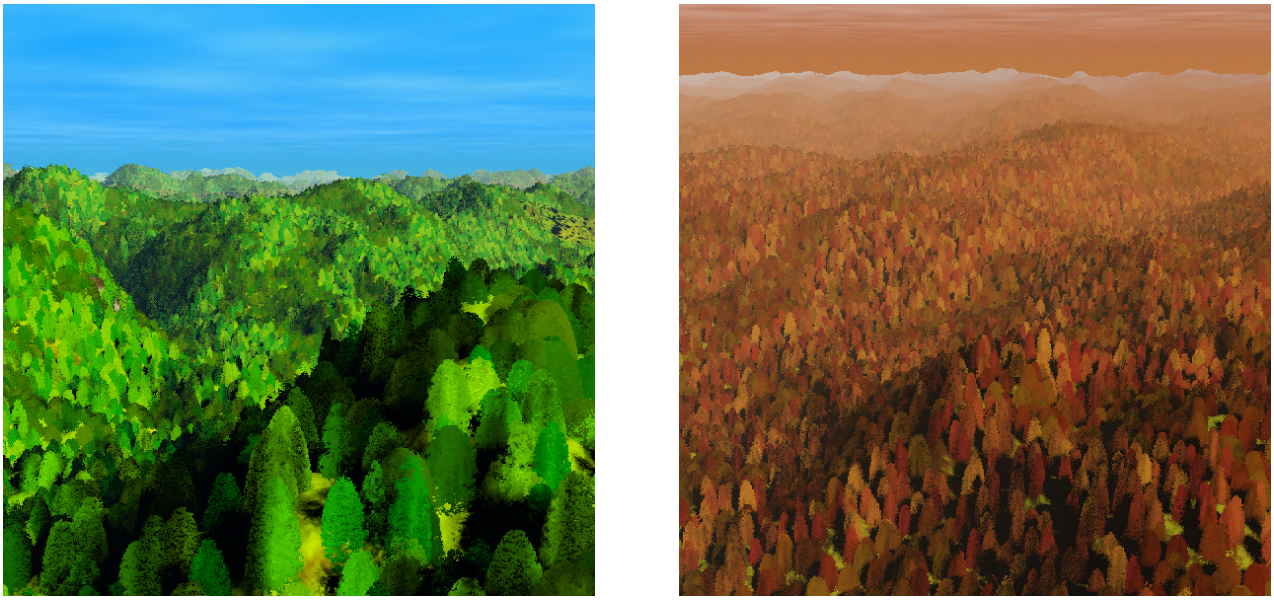


Figure A.1: Example renders from the core version.

A.2 Example Renders of Water

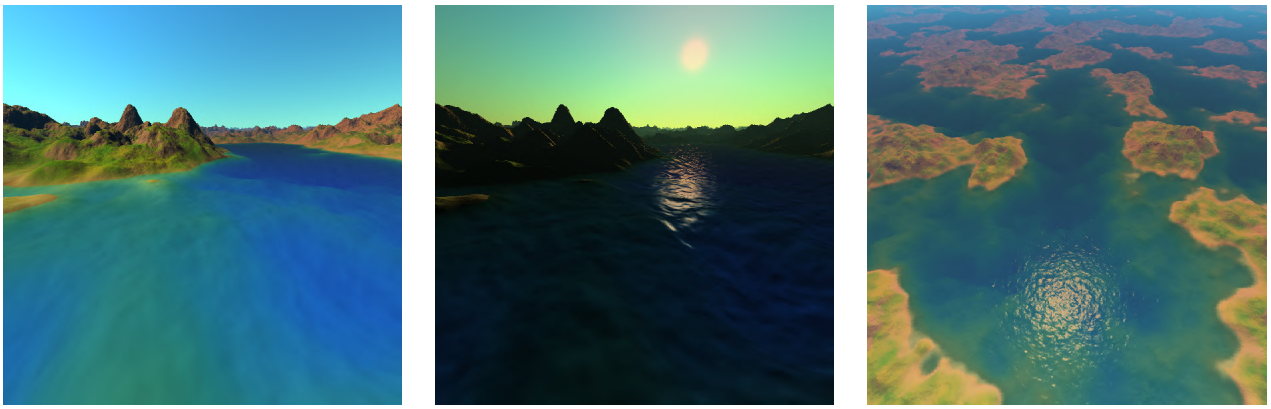


Figure A.2: Example renders of water scenes in my application.

A.3 Example Renders of Physics-Simulated Atmosphere

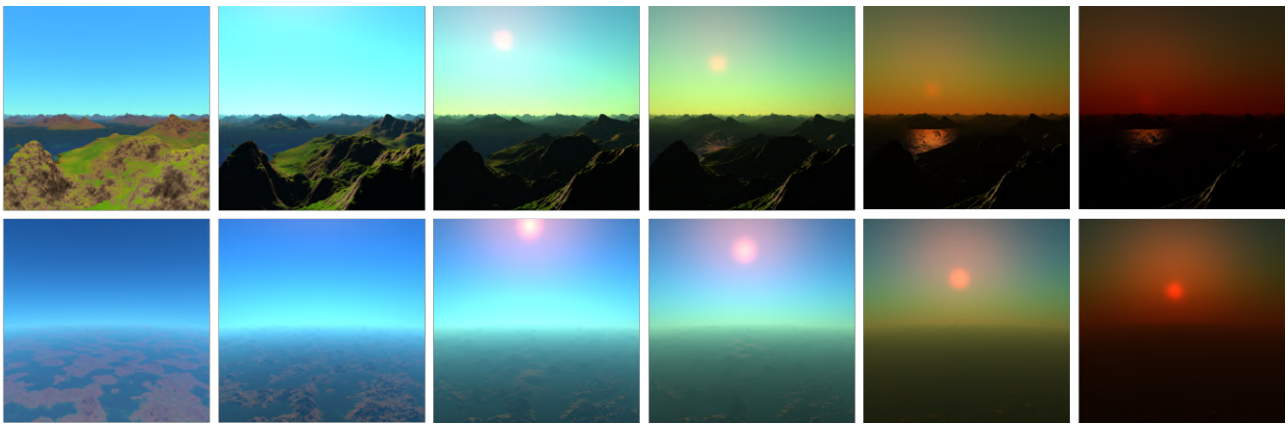


Figure A.3: Displays a series of screenshots with varying sun directions and altitudes: the first row captures lower altitudes, while the second showcases higher altitudes. Progressing from left to right, the sun's angle shifts from being more perpendicular to more oblique relative to the ground, mirroring the natural shift from midday to evening.

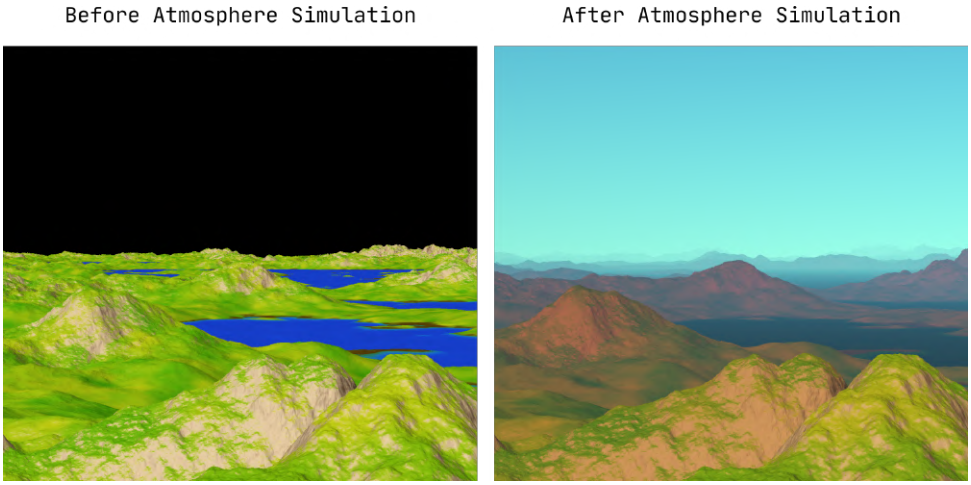


Figure A.4: Compares renders before and after applying the atmosphere simulation, emphasizing the atmospheric perspective on the terrain. Note how the distant terrain appears more merged with the sky due to this effect.

A.4 Example Renders of Volumetric Clouds

Figure A.5: Example renders of volumetric clouds in my application.

A.5 Example Renders of Procedural Planets

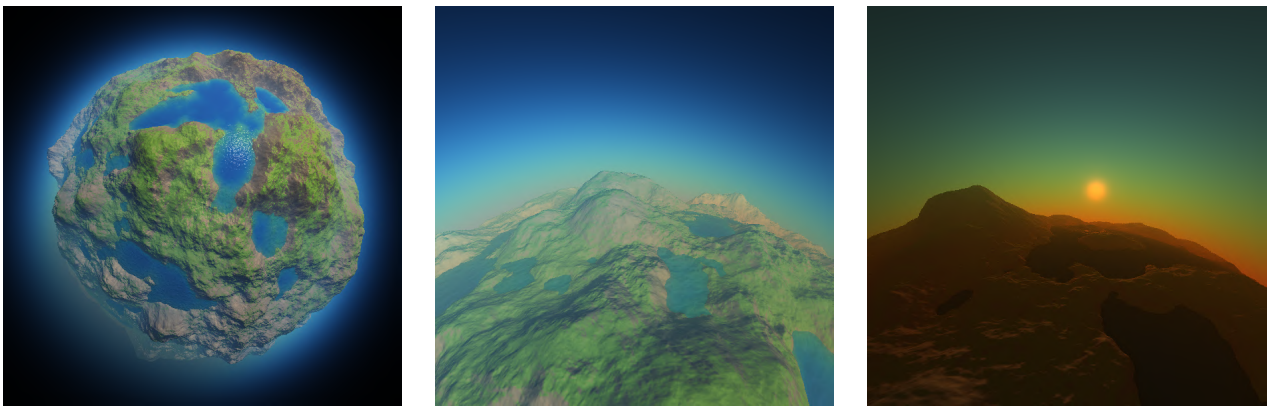


Figure A.6: Example renders of an earth-like procedural planet in my application.

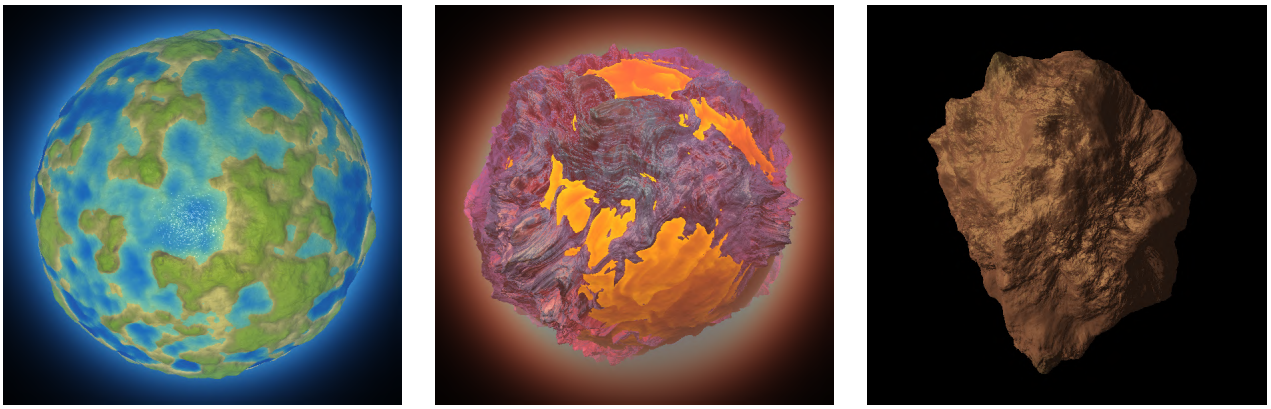


Figure A.7: Example renders of various procedural planets in my application.

B Mathematical Derivations and Formulas

This appendix provides mathematical derivations and formulas used in this project, including the calculations of value noise and its normal vector (Section B.1), the camera ray and the sun ray (Section B.2), a numerical method for calculating the normals of SDFs (Section B.3) and the process of determining the intersections between a ray and a sphere (Section B.4).

B.1 Calculations of Value Noise and Its Normal Vector

B.1.1 Value Noise

Value noise is a fundamental concept in the field of procedural generation, particularly in the construction of fBm (Section C). This section provides a detailed explanation of value noise.

Let's denote the noise function as $N_k(\mathbf{P})$ for a point $\mathbf{P}$ in k-dimensional space. The function operates as follows:

1. **Grid Definition**: Define a grid over the k-dimensional space where each point $\mathbf{p}_i$ on the grid has an associated random value v_i. The distribution of v_i is usually uniform.
2. **Value Assignment**: Assign a random value v_i to each grid point $\mathbf{p}_i$. These values are generated using a pseudo-random number generator and remain constant for a given seed.
3. **Interpolation**: For a point $\mathbf{P}$ not on the grid, identify the surrounding grid points $\{\mathbf{P_i}\}$ and interpolate the values $\{v_i\}$ at these points to calculate $N_k(\mathbf{P})$. The interpolation could be linear, cubic, or use other smoothing techniques, depending on the desired smoothness of the noise.

$$N(\mathbf{P}) = \text{Interpolate}(\{v_i\}, \{\mathbf{P}_i\}, \mathbf{P}) \tag{B.1}$$

B.1.2 Implementation of 2D Value Noise

This subsection describes the method used in my project to compute 2D value noise and its normal for a given point $\mathbf{P}$ in 2D space. This involves determining the noise value at point $\mathbf{P}$ by interpolating between values at surrounding grid points based on the coordinates of $\mathbf{P}$. Implementation of the 3D value noise and its normal is similar but with an additional interpolation step.

For a given point $\mathbf{P}$, first identify the surrounding grid points $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, and $\mathbf{D}$. These points are the corners of the grid cell that encloses point $\mathbf{P}$. Specifically:

$$\begin{aligned}\mathbf{A} &= (\lfloor x \rfloor, \lfloor y \rfloor)\\ \mathbf{B} &= (\lceil x \rceil, \lfloor y \rfloor)\\ \mathbf{C} &= (\lfloor x \rfloor, \lceil y \rceil)\\ \mathbf{D} &= (\lceil x \rceil, \lceil y \rceil)\end{aligned}$$

Using the fractional parts of x and y, denoted as $f_x = x - \lfloor x \rfloor$ and $f_y = y - \lfloor y \rfloor$, perform bilinear interpolation. The smoothstep function s, which is commonly used for smoother transitions, is applied to f_x and f_y. The interpolation process involves:

- Calculating intermediate values $\mathbf{I_1}$ and $\mathbf{I_2}$ based on f_x:

$$\begin{aligned} \mathbf{I_1} &= \mathbf{A} + s(f_x) \times (\mathbf{B} - \mathbf{A}) \\ \mathbf{I_2} &= \mathbf{C} + s(f_x) \times (\mathbf{D} - \mathbf{C}) \end{aligned} \tag{B.2}$$

- Final interpolation between $\mathbf{I_1}$ and $\mathbf{I_2}$ using f_y:

$$N_2(\mathbf{P}) = \mathbf{I_1} + s(f_y) \times (\mathbf{I_2} - \mathbf{I_1}) \tag{B.3}$$

This can be expanded to:

$$N_2(\mathbf{P}) = \mathbf{A} + s(f_x)(\mathbf{B} - \mathbf{A}) + s(f_y)(\mathbf{C} - \mathbf{A}) + s(f_x)s(f_y)(\mathbf{D} + \mathbf{A} - \mathbf{B} - \mathbf{C}) \tag{B.4}$$

Computing the Gradient (Normal)

The normal vector at $\mathbf{P}$, denoted $N_2'(\mathbf{P})$, is crucial for many applications such as lighting calculations. It is derived by taking the cross product of the partial derivatives with respect to x and y of the noise function:

$$\begin{aligned} \frac{\partial}{\partial x} N_2(\mathbf{P}) &= s'(f_x)(\mathbf{B} - \mathbf{A}) + s'(f_x)s(f_y)(\mathbf{D} + \mathbf{A} - \mathbf{B} - \mathbf{C}) \\ \frac{\partial}{\partial y} N_2(\mathbf{P}) &= s'(f_y)(\mathbf{C} - \mathbf{A}) + s(f_x)s'(f_y)(\mathbf{D} + A - \mathbf{B} - \mathbf{C}) \end{aligned} \tag{B.5}$$

Using these gradients, the normal $N_2'(\mathbf{P})$ is computed as:

$$N_2'(\mathbf{P}) = \begin{bmatrix} -\frac{\partial}{\partial x} N_2(\mathbf{P}) \\ 1 \\ -\frac{\partial}{\partial y} N_2(\mathbf{P}) \end{bmatrix} \tag{B.6}$$

B.2 Calculations of the Camera Ray and the Sun Ray

This section details the calculations of the camera ray and the sun ray, which are the initial steps in the fragment shader execution process described in Section 3.1.2.

B.2.1 Calculating the Camera Ray

For each pixel, the process to calculate the camera ray (defined in Section 2.2) involves transforming screen coordinates to a normalized space, and then determining the corresponding direction in the world space.

Screen to normalized space To make the rendering resolution-independent, each pixel's screen coordinates are converted to a normalized space. Let $\mathbf{P}$ be the screen coordinates and $\mathbf{R}$ be the resolution. The normalized coordinates $\mathbf{N}$ are given by:

$$\mathbf{N} = \frac{2 \cdot \mathbf{P} - \mathbf{R}}{\min(\mathbf{R}.x, \mathbf{R}.y)} \tag{B.7}$$

World position calculation To find the world position for each pixel, the camera parameters are used. Let $\mathbf{C}$ be the camera position, $\mathbf{f}, \mathbf{r}, \mathbf{u}$ be the forward, right and up directions respectively, and f be the focal length. Here, f is the distance in world coordinates between the camera $\mathbf{C}$ and the projection plane, influencing the field of view. The world position $\mathbf{W}$ is computed as:

$$\mathbf{W} = \mathbf{C} + \text{normalize}(\mathbf{f}) \cdot f + \mathbf{N}.x \cdot \text{normalize}(\mathbf{r}) + \mathbf{N}.y \cdot \text{normalize}(\mathbf{u}) \tag{B.8}$$

Deriving the camera ray The direction of the camera ray $\mathbf{r_c}$ is the normalized vector from the camera position to the world position of the pixel. It is expressed as:

$$\mathbf{r_c} = \text{normalize}(\mathbf{W} - \mathbf{C}) \tag{B.9}$$

These steps establish the direction of each ray from the camera to the pixel in world space, setting the foundation for ray marching.

B.2.2 Calculating the Sun Ray

In this project, the **sun ray**, $\mathbf{r_s}$, is the normalized vector from a point in the world, towards the sun. The sun is treated as a directional light source due to its immense distance from Earth, implying that the sun rays are parallel at every point.

The sun ray's direction is represented using spherical coordinates, which offer a convenient method to define the direction in 3D space using only two parameters: azimuth (ϕ) and elevation (θ). These parameters are adjustable in the UI, allowing run-time changes to the sun ray. $\mathbf{r_s}$ is computed as follows:

$$\mathbf{r_s} = \text{normalize}\left(\begin{bmatrix}\sin(\theta)\cos(\phi)\\ \cos(\theta)\\ \sin(\theta)\sin(\phi)\end{bmatrix}\right) \tag{B.10}$$

This vector is critical for lighting calculations in the rendering process.

B.3 Numerical Method for Calculating Normals of SDFs

In this project, I utilized a numerical method to calculate the normal of a SDF, as discussed in Section 3.3.4. This method [43] involves sampling four points arranged in a tetrahedron with vertices $\mathbf{k}_0 = (1, -1, -1)$, $\mathbf{k}_1 = (-1, -1, 1)$, $\mathbf{k}_2 = (-1, 1, -1)$, and $\mathbf{k}_3 = (1, 1, 1)$. The normal at point $\mathbf{P}$ is then computed as:

$$\mathbf{n}(\mathbf{P}) = \sum_i \text{SDF}(\mathbf{P} + \lambda \mathbf{k}_i)\mathbf{k}_i \tag{B.11}$$

B.4 Intersection of a Ray and a Sphere

This section outlines the method for finding intersections between a ray and a sphere, a key process for the water and the atmosphere in procedural planets, as discussed in Section 3.7.4.

A point $\mathbf{P}$ on the ray is described by $\mathbf{P} = \mathbf{O} + t\mathbf{d}$, where t is a scalar representing the distance along $\mathbf{d}$ from the ray origin $\mathbf{O}$. A sphere with center $\mathbf{C}$ and radius r, is described by $(\mathbf{P} - \mathbf{C})^2 = r^2$.

Substitute the ray equation into the sphere's equation:

$$(\mathbf{O} + t\mathbf{d} - \mathbf{C})^2 = r^2 \tag{B.12}$$

Expanding this and using the fact that $\mathbf{d}$ is normalized ($\mathbf{d} \cdot \mathbf{d} = 1$):

$$(\mathbf{O} - \mathbf{C} + t\mathbf{d})^2 = r^2 \tag{B.13}$$

$$(\mathbf{O} - \mathbf{C})^2 + 2t(\mathbf{O} - \mathbf{C}) \cdot \mathbf{d} + t^2 = r^2 \tag{B.14}$$

Let $\mathbf{v} = \mathbf{O} - \mathbf{C}$:

$$\mathbf{v}^2 + 2t\mathbf{v} \cdot \mathbf{d} + t^2 = r^2 \tag{B.15}$$

This simplifies to a quadratic in t:

$$t^2 + 2t(\mathbf{v} \cdot \mathbf{d}) + (\mathbf{v}^2 - r^2) = 0 \tag{B.16}$$

The quadratic formula gives:

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{B.17}$$

where $a = 1$, $b = 2\mathbf{v} \cdot \mathbf{d}$, and $c = \mathbf{v}^2 - r^2$. The discriminant ($b^2 - 4ac$) reveals:

- **Positive**: Two intersection points.
- **Zero**: One point (tangent).
- **Negative**: No intersection.

C Implementation of fBm

As discussed in Section 2.1.2, we can construct fBm using fractal noise. The function $\text{fBm}_k(\mathbf{x}, n)$ is evaluated as outlined in Algorithm 3. The specifics of the value noise implementation are detailed in Appendix B.1. The gradient, and therefore the normals, of the fBm can be derived by summing the normals of the value noise, converted to world space, from each layer.

Algorithm 3 Compute fBm Value at a Point

```
function fBm(x, n, initial_params, adjustment_factors)
    fBm_value ← 0
    layer_params ← initial_params
    for i = 0 to n − 1 do
        fBm_value ← fBm_value + ComputeLayerValue(x, layer_params)
        layer_params ← UpdateLayerParams(layer_params, adjustment_factors)
    end for
    return fBm_value
end function

function UpdateLayerParams(current_params, adjustment_factors)
    f_i, a_i, R_i, o_i ← current_params
    α, β, R, o ← adjustment_factors
    return (f_i · α, a_i · β, RR_i, o + o_i)
end function

function ComputeLayerValue(x, layer_params)
    f_i, a_i, R_i, o_i ← layer_params
    transformed_x ← f_i · R_i x + o_i
    return a_i · ValueNoise(transformed_x)
end function
```

D Basic Implementations of the Atmosphere and the Clouds

In the core version of the application, I used basic implementations for modeling the atmosphere and clouds. Later, these were extended with the introduction of volumetric clouds (Section 3.6) and a physics-simulated atmosphere (Section 3.5).

D.1 Basic Implementation of the Atmosphere

In the core version, I approximated the sky color using a simple gradient – a linear interpolation between a color at the bottom and another at the top of the screen. While this method does not account for sun direction or display the varied color nuances of a real sky, it can still yield reasonably natural results if the colors are chosen carefully, as demonstrated in Figure D.1.

I modeled atmospheric effects by blending the fragment color with a factor τ, calculated as:

$$\tau = \exp(-\lambda \cdot d) \tag{D.1}$$

Here, λ controls the effect strength, and d is the distance from the camera to the scene intersection.

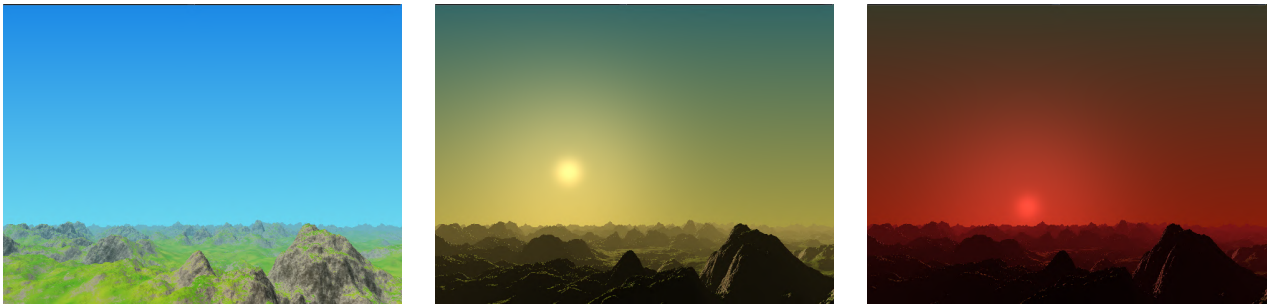


Figure D.1: Example renders of the basic implementation of the atmosphere.

D.2 Basic Implementation of the Clouds

In the core version, I simulated clouds on a plane, utilizing a 2D fBm to model their density. If a ray intersects the plane, the density value is used to blend between the sky color and the cloud color. The resulting visuals often appear flat and can seem distorted when viewed from shallow angles, as illustrated in Figure D.2.

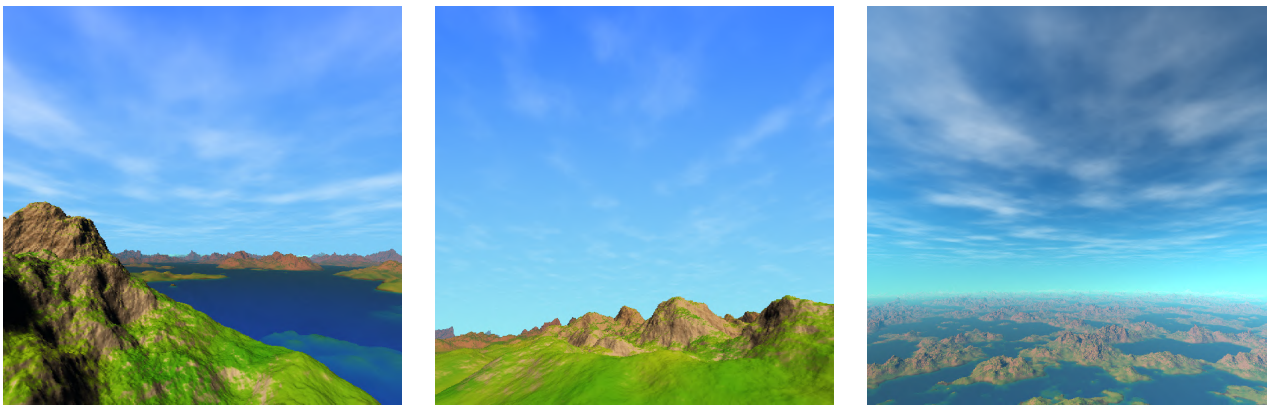


Figure D.2: Example renders of the basic implementation of the clouds.

E Implementation Details for Input Controls

This appendix provides the implementation details for the camera controls and callback system in my project.

E.1 Camera Controls

In this project, the scene features an infinitely expansive terrain, making traditional CAD camera controls like turn-table or trackball models unsuitable. A **first-person game-like camera control model** [35] was adopted for more intuitive navigation in such an extensive environment. The camera movement is controlled by the WASD and EQ keys, which allow movement along the three axes of the camera's coordinate system. The left mouse button drag controls the forward direction of the camera, while the scroll wheel adjusts the focal length. Right mouse dragging enables movement in the plane perpendicular to the forward direction.

Moreover, I added **trackball** camera controls [36] for navigating planets in my planet extension (Section 3.7), and included an option in the UI to switch control modes, as shown in Figure E.1.

Camera parameters, including position and rotation, are first updated on the CPU using GLFW, and then passed as uniforms to the fragment shader. For detailed adjustments, the precise position and rotation can be fine-tuned through the UI. Figure E.1 displays the camera panel in the UI.

This control model enables efficient navigation through the extensive virtual world, proving especially useful for debugging and locating ideal camera angles for rendering.

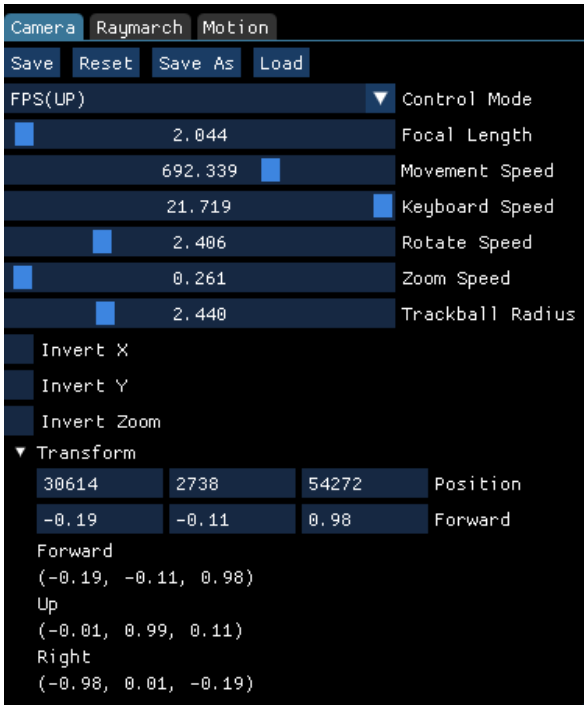


Figure E.1: The camera panel in the UI, featuring adjustable settings and camera transforms.

E.2 Callback System

To address GLFW's limitation of handling only one callback per event, a centralized callback management system was implemented. This system stores and triggers user-defined callback functions for various input events. It activates the appropriate callbacks in response to specific events, thereby facilitating multiple reactions to a single input. This structure significantly enhances the flexibility and complexity of input management within the application.

F Implementation Details for the UI

This appendix outlines the implementation details of the application's UI and the save and load system.

F.1 Hierarchical Structure

The UI of the application was implemented following an Object-Oriented approach and a hierarchical structure, represented in Figure F.1.

The **App** is the singleton entity at the top level, containing multiple **panels**. Each panel inherits from a common abstract class, and comprises several **properties**. These properties are again derived from an abstract class and are responsible for managing individual or groups of parameters. The parameters correspond to uniforms on the GPU, facilitating real-time updates. UI interactions trigger updates to the corresponding uniforms in the GPU.

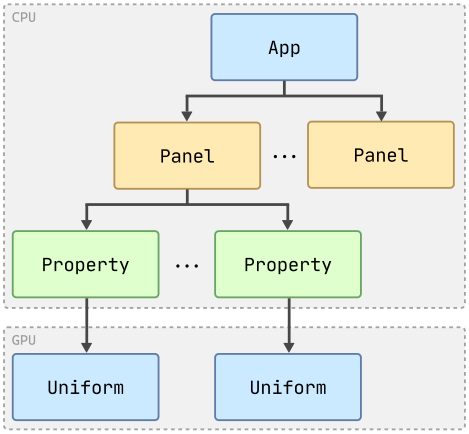


Figure F.1: Outlines the hierarchical structure of the UI.

The hierarchy in implementation directly corresponds to the visual structure of the UI, as illustrated in Figure F.2.

F.2 Composite Pattern

In enhancing the UI's flexibility and organizational structure, the **Composite Design Pattern** plays a crucial role. This design pattern allows for treating individual objects and compositions of objects uniformly. In the context of the app's UI:

- **GroupProperty**: Implemented as a single property, it is a collapsible group containing multiple properties.
- **TabPanel**: Implemented as a singular panel, it contains multiple sub-panels organized as tabs.

Both elements allows for efficient space usage and better categorization, as demonstrated in Figure F.3.

F.3 Save and Load

The UI supports saving and loading of parameters and layout preferences.

Selection of JSON format I chose the JSON format for the app's saving and loading because of its compatibility with various data types, its hierarchical structure that mirrors the app's design, and its human readability.

Recursive Implementation The save and load functionality is implemented in a recursive manner. This design choice simplifies the process and ensures consistency across different levels of the UI. Each class in the application, including the app itself, every panel, and every property, implements save_to_json and load_from_json methods. This recursive approach mirrors the hierarchical nature of the UI, enabling seamless serialization and deserialization of parameters.

Figure F.2: Demonstrates the visual structure of the UI.

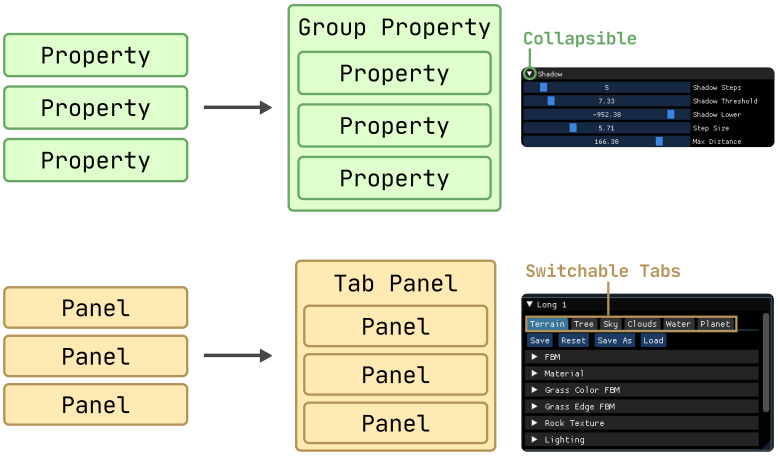


Figure F.3: Demonstrates GroupProperty and TabPanel. GroupProperty allows several properties to be organized into a collapsible group. TabPanel allows several panels to be organized as tabs in a single panel.

Scope of Save and Load One of the benefits of this hierarchical and recursive implementation is the flexibility it offers in terms of scope. It is possible to save and load configurations either globally or at the individual panel level. This flexibility is reflected in the UI, where buttons are provided in the menu bar for global operations, and within each panel for more localized control, as demonstrated in Figure F.4.

Example JSON Structure A typical JSON file structure for saving the app's parameters resembles the following:

```
{
    "Panel 1": {
        "Param X": {
            "value a": 10,
            "value b": 12
        },
        "Param Y": {
            "value": 1.2
```

```
        }
    },
    "Panel 2": {
        "Param Z": {
            "value": true
        }
    }
}
```

F.4 Layout Preferences

In addition to parameter values, layout preferences such as the sizes and positions of panels are also managed through the save and load system. As shown in Figure F.4, the menu bar includes options for saving and loading these layout configurations, further enhancing the app's user experience.

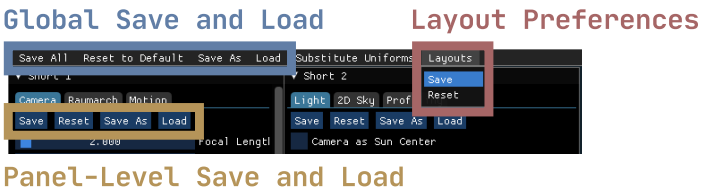


Figure F.4: Demonstrates the UI of save and load functionalities.

G Source Code for Triplanar Normal Mapping

This appendix provides the code for triplanar normal mapping, which is utilized in the illumination of procedural planets as discussed in Section 3.7.3.

```
vec3 normal_blend(
    in vec3 _direction,
    in vec3 _xnormal,
    in vec3 _ynormal,
    in vec3 _znormal
){
    vec3 weights = pow(abs(_direction), vec3(iTriplanarMappingSharpness));
    weights = weights / (weights.x + weights.y + weights.z);

    // whiteout blending
    _xnormal.xz += _direction.zy;
    _xnormal.y  =  _direction.x * abs(_xnormal.y);
    _ynormal.xz += _direction.xz;
    _ynormal.y  =  _direction.y * abs(_ynormal.y);
    _znormal.xz += _direction.xy;
    _znormal.y  =  _direction.z * abs(_znormal.y);

    vec3 normal_world = normalize(
        _xnormal.yzx * weights.x +
        _ynormal.xyz * weights.y +
        _znormal.xzy * weights.z
    );

    return normal_world;
}
```

H Proposal

Project Proposal

Procedural Generation of Complex Terrain With Implicit Representation and Raymarching

2395B

May 11, 2024

1 Introduction

From the intricate patterns on tree barks to the vast expanse of terrains and cloud-filled skies, nature's canvas is a marvel of self-similar structures and complex patterns. Attempting to replicate these structures digitally, with precision and realism, has been a long-standing challenge and a topic of keen interest for computer scientists, artists, and game developers alike.

Procedural terrain generation has come a long way since its inception. The late 1970s saw the introduction of fractals, capturing nature's self-similarity. In the 1980s, Perlin noise emerged, laying the groundwork for various procedural methodologies. The 1990s built on this, employing fBM to generate heightmaps, and later, erosion simulations added more realism. The 2010s took things into the third dimension, with methods like marching cubes to extract a 3D mesh from volumetric data. Today, tools like Generative Adversarial Networks (GANs) have been employed to make even more realistic terrains.

Procedurally generated terrains often rely on 3D meshes, which are subsequently rendered using standard rasterization within the graphics pipeline. While effective, these mesh-based techniques can be cumbersome for vast terrains with intricate details due to their sizable memory footprint and fixed resolution. The real-time generation further poses challenges due to the computationally intensive mesh extraction process.

Implicit representations present an alternative, encapsulating detailed, expansive terrains within a singular mathematical function. This approach boasts minimal storage overhead and infinite resolution without the need for mesh extraction. While such representations can't be directly rasterized in the conventional sense, they align well with raymarching—a technique akin to raytracing that iteratively steps through rays to discern intersections. Yet, despite its merits, raymarching implicit terrains remain less explored. This project endeavors to uncover the capabilities and nuances of raymarching within the realm of implicit terrain representations.

2 Description

The core goal of this project is to procedurally generate realistic natural terrains with details and cloudscapes and then render them using raymarching in real time.

2.1 Terrain Generation Using fBM

Fractional Brownian Motion (fBM) is known for its randomness and self-similarities, which can be observed in numerous natural phenomena such as terrains, clouds, and tree barks. Given this, fBM serves as a valuable tool to generate heightfields and density maps for terrains and clouds.

Constructing an fBM involves invoking deterministic randomness through a noise function and establishing self-similarity with the data. This process initiates with a foundational noise signal, augmented with progressively detailed noise signals [7].

The core of the project will utilise fBM to create terrain heightfields while exploring various noise functions such as perlin, simplex and worley noises.

2.2 Terrain Details and Clouds

To add details to the landscape, domain repetition will be employed. This method enables the duplication of infinitely many SDF primitives, such as ellipsoids representing trees, through a single function. By computing a distinct ID for each instance, variations among the duplicates can be introduced [6].

Additionally, for cloud generation and rendering, density maps will be constructed using fBM using 3D noise and then rendered via volumetric raymarching. To compute lighting, Beer's law and the Henyey-Greenstein phase function will be employed [3].

2.3 Raymarching for Rendering

Raymarching, a technique related to raytracing, has been selected as the primary rendering method for this project. It determines intersections by methodically advancing light rays, making it especially suitable for implicit surfaces like Signed Distance Functions (SDF). Raymarching heightfields can be straightforward using a basic approach that takes fixed steps, halting when the ray is first positioned beneath the heightfield. For a more precise intersection point determination, bi-linear interpolation can be utilized.

The rendering will be complemented with Phong shading, and procedural albedo coloring will be derived from attributes like the normals at intersection points. Additionally, the project will introduce soft shadows, calculated from the minimal vertical distance between the terrain surface and the shadow ray, starting at the intersection and tracing the path of the shadow ray.

2.4 Desktop Application

Although online platforms like `shadertoy.com` provide the capability to perform raymarching and procedural generation in shaders, a dedicated desktop application will be developed. This software will feature an intuitive GUI, allowing the candidate to dynamically modify raymarching and terrain generation hyperparameters. It will support saving and loading of hyperparameters in an easily editable plain-text format, such as JSON. The application will also include camera controls for panning, rotating, and zooming, simplifying the process of navigating the scene to identify the optimal rendering angle. Designed primarily for personal development, this tool could be beneficial for knowledgable users as well.

3 Success Criteria

The project's success criteria are outlined as follows:

- Achieve real-time raymarched rendering of natural terrains, encompassing shadows, shading, and colouring. *Screen recordings showcasing the rendered outcomes will be submitted to the department to highlight the real-time performance capabilities.*
- Attain realistic detailing in the terrains, including features like foliage and rocks, complemented by an authentic backdrop of clouds and sky.
- Develop a desktop application that offers camera controls, allows real-time hyperparameter adjustments, and supports saving and loading of hyperparameters in an accessible plain-text format, such as JSON.

4 Possible Extensions

4.1 Complex Terrain

The primary focus of these extensions is to augment the intricacy of the terrain, incorporating unique features such as caves and overhangs. Heightfield representations inherently have limitations, as they define a singular height for each point, making such additions unfeasible.

To overcome this constraint, the following 2 extensions are proposed:

1. **Implicitization**: Convert the heightfield into an implicit function. Subsequently, enhance this implicit terrain using sculpting primitives [4].
2. **3D Scalar Fields**: Instead of solely depending on 2D scalar heightfields, this approach involves generating 3D scalar fields using fBM [2]. While typically this method is followed by the marching cubes algorithm to derive a mesh, in this project, the 3D scalar fields will be treated as implicit representations for direct rendering.

Furthermore, an additional extension will involve the integration of raymarched fractals, such as the Mandelbrot Set, Julia Set, and The Kaleidoscopic IFS fractals, to infuse distinct and visually captivating non-natural elements [1].

4.2 Performance Optimisation

As an extension, this project will incorporate a UI in the destop app to evaluate and display performance metrics such as frame rate.

This project will try different methods to enhance raymarching efficiency. The methods are early ray termination, and adaptive step sizes techniques such as cone stepping.

4.3 Integrate with Traditional Rasterization

As an extension, the project aims to integrate raymarching within the standard graphics pipeline. This will permit a hybrid rendering model, where for example the background features raymarched terrains, and the foreground exhibits rasterized character meshes [5].

4.4 Interactive Sculpting Tool

As an extension to the project, the desktop application will incorporate an interactive editor. This feature will enable users to select from a range of implicit primitives, such as spheres and cubes, to serve as sculpting tools. Utilizing mouse movements, users can sculpt the procedural terrain in real time either additively or reductively based on their mouse inputs. This functionality is similar to the sculpting capabilities found in modeling software like Blender and Maya. This extension will leverage the inherent advantages of implicit representations for smooth blending and efficient boolean operations.

5 Evaluation

The effectiveness and efficiency of this project will be assessed using a combination of objective and subjective metrics:

- **Frame Rate**: The frames per second (FPS) for various raymarching techniques and hyperparameters for procedural generation will be recorded and compared.

  Given the nature of implicit representations utilized in this project, traditional metrics such as memory usage and load times may not be directly applicable. This is because implicit representations are implemented as code within the fragment shader, thus having minimal associated data.

- **Scalability**: The system's performance will be gauged in response to increasing terrain size and complexity, as well as rising render resolutions. Key considerations will include whether the system can uphold consistent performance levels and maintain visual quality as these hyperparameters evolve.

- **Subjective Analysis**: In order to find the set of hyperparameters that balance performance and quality, participants will be asked to assess both images and videos of terrains generated by the proposed method using different hyperparameters. Natural terrains will be evaluated for their naturalness and visual appeal. In contrast, unnatural terrains crafted using fractal extensions will only be judged on their visual appeal.

It's relevant to note that the desktop application associated with this project will not undergo an usability evaluation. The application is primarily designed as an internal tool, facilitating easier render creation and performance monitoring for the candidate.

6 Starting Point

The project will leverage several pre-existing libraries:

- **OpenGL:** To establish the graphics pipeline.
- **GLFW:** For desktop app window management.
- **Dear ImGui:** For desktop app UI development.
- **CMake:** To streamline project structure and facilitate compilation with the above libraries.

Beyond these libraries, all other aspects of the project will be developed from the ground up, with no pre-existing code being used.

My background includes:

- Familiarity with OpenGL wrapper libraries from the "Introduction to Graphics" course and a past internship, though direct experience with OpenGL itself is limited.
- Practical experience with Dear ImGui from a previous internship.
- Some exposure to HLSL through the "Introduction to Graphics" course.
- C++ experience gained both in the "Programming in C and C++" course and a past internship.

My curiosity about raymarching implicit surfaces was piqued by YouTube videos from Inigo Quilez. While the subject remains new to me, I have had a brief hands-on experience in the summer experimenting with a shader on shadertoy.com that raymarched SDFs of spheres.

7 Resources

I will be using a personal laptop with the following specs:

Model: Lenovo Legion 5 15ARH05H
GPU: NVIDIA GeForce RTX 2060, 6GB VRAM
CPU: AMD Ryzen 7 4800H with Radeon Graphics 2.90 GHz
RAM: 16GB
Disk: 1.5TB
OS: Windows 10, 64 bit

I have a backup personal machine with the following specs:

Model: Macbook Pro 2015
GPU: Intel Iris Graphics 6100
CPU: Intel I5 2.7 GHz
RAM: 8GB
Disk: 512GB
OS: macOS, Windows (BootCamp)

I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.

I intend to use Git as my version control software and store the work in progress on my machine with daily backups to GitHub.

8 Timetable

Slot	Work	Milestone
	Michaelmas	
	Project Proposal Deadline: 16/10/2023	

Week 3-4 (19/10/2023 01/11/2023)	○ Set up libraries and development environments. ○ Generate heightfields of terrains using fBM and visualize them via simple fixed step raymarching. ○ Apply phong shading and introduce soft shadows to terrains.	Generate shaded terrain renders.
Week 5 (02/11/2023 08/11/2023)	○ Enhance terrain realism by integrating foliage and other details using domain repetition of SDF primitives.	Generate realistic terrains renders.
Week 6-7 (09/11/2023 22/11/2023)	○ Generate clouds using 3D noises and fBM; compute lighting via Beer's law and Henyey-Greenstein's phase function.	Render realistic cloudscape atop terrains.
Week 8 (23/11/2023 29/11/2023)	○ Develop a basic, but expandable, UI for hyperparameter adjustments. ○ Introduce functionalities for hyperparameter saving and loading. ○ Facilitate camera controls, including pan, rotate, and zoom. ○ *Limited work to accommodate MVP project deadline on 01/12/2023.*	Functional GUI. **[Achieve success criteria]**.
	Christmas	
Week 1-2 (30/11/2023 13/12/2023)	○ **Holidays**	
Week 3-4 (14/12/2023 27/12/2023)	○ [**Extension 4.1**] Generate terrain 3D scalar maps using 3D noises and fBM. Integrate terraces and establish floor levels.	Generate renders of terrains enriched with caves and terraces.
Week 5 (28/12/2023 03/01/2024)	○ [**Extension 4.3**] Merge raymarching with conventional rasterization via a depth buffer. ○ Write the Introduction and Preparation chapters.	Generate renders of meshes in the foreground and raymarched terrain in the background. Submit the 2 chapters for supervisor review.
Week 6-7 (04/01/2024 17/01/2024)	○ Refine the Introduction and Preparation chapters based on feedback. ○ [**Extension 4.2**] Incorporate a UI to evaluate performance and establish the performance tool *RenderDoc*. ○ Enhance raymarching efficiency using early ray termination and adaptive step sizes.	Report contrasting performance across various raymarching techniques and hyperparameters.
	Lent	
	Progress Report Deadline: 02/02/2024	

Week 1-2 (18/01/2024 31/01/2024)	○ [**Extension 4.4**] Implement an interactive sculpting tool. ○ Prepare Progress Report and presentation.	Produce screen recordings of real time sculpting.
Week 3-4 (01/02/2024 14/02/2024)	○ [**Extension 4.1**] Raymarch mathematical fractals and integrate them into terrains, adding unique features.	Generate renders of unnatural terrains.
Week 5-7 (15/02/2024 06/03/2024)	○ [**Extension 4.1**] Transform heightmaps to implicit forms and enhance terrains with intricate 3D features such as caves, cliffs, and arches using sculpting primitives.	Generate renders of terrains with distinctive topographical details.
Week 8 (07/03/2024 13/03/2024)	○ **Slack period** to accommodate any unfinished work. ○ *Limited work to accommodate Extended Reality project deadline on 14/03/2024.*	
	Easter Break	
Week 1-2 (14/03/2024 27/03/2024)	○ Write the Implementation chapter.	Submit the Implementation chapter for supervisor review.
Week 3 (28/03/2024 03/04/2024)	○ **Holidays**	
Week 4 (04/04/2024 10/04/2024)	○ Revise the Implementation chapter based on feedback. ○ Write the Evaluation and Conclusion chapters.	Submit the full draft to supervisor for review.
Week 5-6 (11/04/2024 24/04/2024)	○ Implement feedback on the dissertation and refine the code repository.	Submit the finalized dissertation to supervisor.
	Easter Term Dissertation and Source Code Deadline: 10/05/2024	
Week 1-2 (25/04/2024 08/05/2024)	○ Implement final feedback on the dissertation.	Finalize and submit the dissertation and source code.

Table 1: Schedule for the Part II project

References

[1] M. H. Christensen. *Distance Estimated 3D Fractals*. 2011. URL: http://blog.hvidtfeldts.net/index.php/2011/06/distance-estimated-3d-fractals-part-i/.

[2] R. Geiss. "Generating Complex Procedural Terrains Using the GPU". In: *GPU Gems 3*. n.d. Chap. 1. URL: https://developer.nvidia.com/gpugems/gpugems3/part-i-geometry/chapter-1-generating-complex-procedural-terrains-using-gpu.

[3] F. Haggstrom. "Real-time Rendering of Volumetric Clouds". In: (2018). URL: https://www.diva-portal.org/smash/get/diva2:1223894/FULLTEXT01.pdf.

[4] A. Paris et al. "Terrain Amplification with Implicit 3D Features". In: *ACM Transactions on Graphics* 38.5 (2019), Article 147. DOI: https://doi.org/10.1145/3342765.

[5] I. Quilez. *Depth Buffer with Raymarching*. n.d. URL: https://iquilezles.org/articles/raypolys.

[6] I. Quilez. *Domain Repetition*. n.d. URL: https://iquilezles.org/articles/sdfrepetition.

[7] I. Quilez. *FBM*. n.d. URL: https://iquilezles.org/articles/fbm.