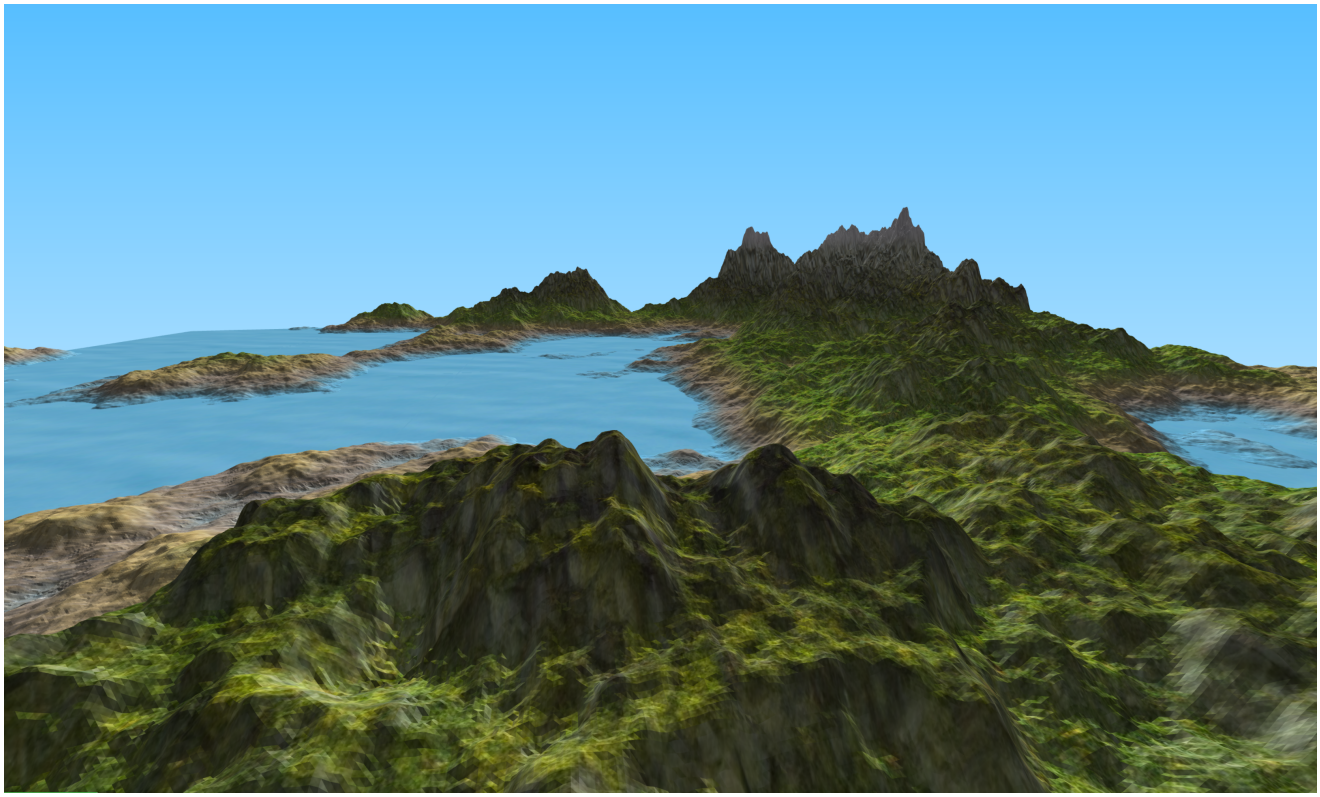
Parametrized Procedural Terrain Generation



Master Project Report
16.06.2024

by Christian Skorski, Kalliopi Papadaki, and Konstantin Moser

Supervisors:
Prof. Dr. Renato Pajarola
Julian Croci

Visualization and MultiMedia Lab
Department of Informatics
University of Zürich

University of Zurich[UZH]

VISUALIZATIONANDMULTIMEDIALAB

Abstract

This report details the development of a web-based interactive graphical application for generating terrains procedurally. This includes the development or integration of diverse generation and manipulation algorithms, each distinct in its nature. Many of our generators rely on algorithms not necessarily made specifically for terrain generation, such as Perlin and Simplex noise, Diamond-Square, Lindenmayer Systems, and Voronoi tessellation. Key to this project is the user-friendly interface allowing users to effortlessly create, modify, and visualize different terrain types. This report delves into the technology stack utilized and explores the technical solution by providing an in-depth description of the developed algorithms and data structures used. We then go over the implementation details, focusing on code organization. Future enhancements will focus on refining the realism of generated terrains, expanding user control options and general performance. This application may serve as a valuable tool for conducting sensitivity analysis of numerical weather models, enabling users to manipulate and observe how changes in terrain affect atmospheric simulations.

Contents

1 Introduction

Computer science and its intersection with visual media has seen a remarkable evolution in the automatic generation of 3D terrain. Utilized across a spectrum of applications, from video games and movies to image creation, terrain generation has primarily been geared towards aesthetic visualizations. Techniques like procedural generation, fractals, and machine learning have been pushing the boundaries of visual realism. However, the utility of these technologies extends beyond visual appeal, particularly when it comes to scientific simulations and analyses.

The significance of terrain generation takes on a different dimension in scientific contexts. Unlike applications focused on visualisation for entertainment, scientific simulations demand terrain generation that adheres to specific, quantified characteristics rather than aesthetic qualities. This shift in focus is particularly relevant for sensitivity analysis of weather models, in which the terrain's numerical parameterization plays a crucial role.

This Master Project sets its sights on exploring terrain generation from a new perspective: Procedural generation with fine-detailed numerical control. The goal is not just to create terrain that looks accurate and realistic, but also to make it highly adjustable based on defined parameters. By implementing various parameterized terrain generation algorithms, the user is able to shape the terrain to their specific needs. Compared to static methods (manual modeling), procedural terrain generation stands out for its speed, offering practically infinite variations of terrain features from a highly compact function representation. This approach allows to quickly generate high numbers of diverse and realistic terrains. Even though procedural terrains have several benefits compared to static models, designing these can be unintuitive and often complex [1].

The application developed in this project is designed to be user-friendly, featuring a versatile user interface that allows for the dynamic selection and modification of terrain generation methods and parameters. The goal is to enable users to easily generate, visualize, compare, and save different terrains, along with their defining parameters. The interface also assists in the scale estimation of a generated landscape by enabling to set the side length in Km and computing the relative height of the highest peak.

Looking ahead, the potential applications of this project go beyond its immediate scope. The user-friendly procedural generation of terrain could aid future research, especially in assessing how terrain features affect weather models. By allowing the creation of detailed and numerically defined terrains, this project provides a foundation for further studies in the sensitivity analysis of numerical weather models with regard to terrain characteristics.

2 Technical Solution

We have built the project using the following technologies: Docker (Compose), Python Flask, MongoDB, ReactJS, Redux, WebGL2, and TWGL. This tech stack affords robustness, scalability, and efficiency, catering to both backend and frontend needs. Docker guarantees system consistency and scalability, Python Flask manages the server side, and MongoDB caches generated terrain data. In the frontend, ReactJS and Redux enhance interactivity and user experience, while WebGL2 and TWGL support the rendering of generated terrains in 3D, ensuring uncomplicated and high-quality visual output.

2.1 Technology Stack

2.1.1 Docker

Docker [2] ensures consistency and ease of deployment across different machines and operating systems. The setup involves two Dockerfiles, in their respective backend and frontend directories: The backend Dockerfile fetches a lightweight Python 3.10 Docker image and sets the environment within a container, installing necessary Python packages and finally copies our server-side code from the host system and runs it. Similarly, the frontend Dockerfile fetches and instantiates a lightweight Node Docker image, installs frontend dependencies and copies and runs our client-side code.

The two Dockerfiles are linked by a Docker-Compose file that sets up environment variables, links network ports between the containers and the host system, and sets up volumes that allow syncing data between the host and the containers. Additionally, it fetches and instantiates a MongoDB image, which we will describe shortly. A diagram of our current architecture and technology stack can be appreciated in figure 2.1.

Docker is really useful, because it allows to effortlessly deploy our application on any system that also supports Docker. In fact, our team has been working on the project using three different operating systems and two different CPU architectures (x86, ARM) without experiencing any major issues.

2.1.2 Python Flask

We use Python Flask [3] for our server and REST API. Flask enables the creation and management of web routes and requests in a minimal and clean way. Additionally, the application employs Flask extensions such as Flask-PyMongo for database interactions, and Flask-Compress for compressing responses with the Brotli algorithm [4] to reduce network latency and enhance performance. The main advantage of using Python as backend is the ease and speed of development. The sacrifice in computing performance is small to none if using NumPy and other C-based libraries to tackle heavier tasks. The backend primarily handles terrain generation, hydraulic erosion, terrain caching in MongoDB, and vertex normals computation.

2.1.3 MongoDB

MongoDB [5] is employed in the application's backend for terrain caching. It saves generated terrain data, utilizing a hash of the terrain parameters as key. The key is then used to identify pre-existing terrains and allows directly returning them to the frontend instead of regenerating them. This approach allows quickly switching between already generated terrains.

To allow storing larger terrains ($> 1000 \times 1000$) we use Snappy [6], a rapid compression algorithm, in combination with GridFS, "a [MongoDB] specification for storing and retrieving files that exceed the BSON-document size limit of 16 MB" [7].

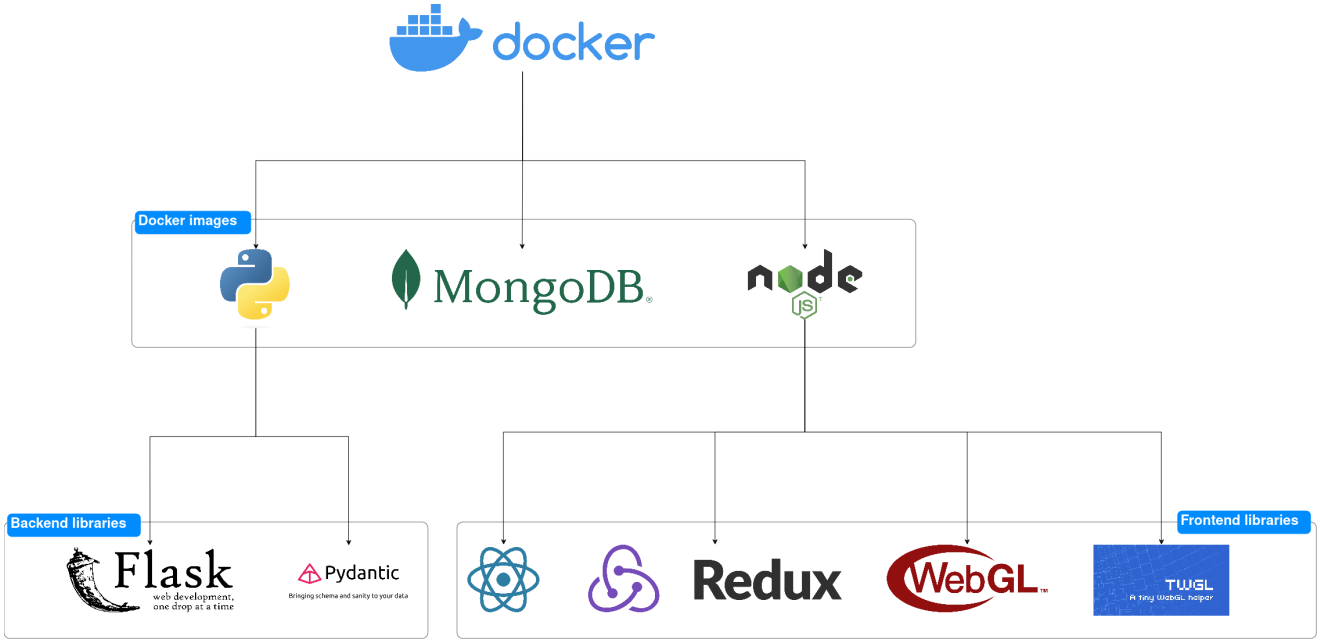


Figure 2.1: The project's architecture and technology stack

2.1.4 ReactJS

The frontend is powered by ReactJS [8]. React enables the application to be single-page, avoiding page refreshes and providing a more seamless and interactive user experience. It also partially simplifies development thanks to its JSX syntax, which allows it to easily combine HTML code with JavaScript/ReactJS elements and values.

2.1.5 Redux

Redux [9] centralizes state management in the frontend, organizing and handling application states across different components and actions. It provides a consistent and simplified approach to data flow and state updates, improving code readability and easing development.

2.1.6 WebGL2 and TWGL

WebGL2 [10] is used for rendering terrain data in 3D. This process is handled by specialized classes that instantiate shader programs and manage the visual representation pipeline. The application allows the user to interactively engage with the terrain by translation, rotation, and zoom, which will be described in more detail in section 2.2.

Tiny WebGL [11] is a minimal library that assists with simplifying WebGL-related tasks. It streamlines the setup and management of the graphics pipeline by greatly reducing WebGL boilerplate code, allowing for faster development and better code readability.

2.1.7 Noise Libraries

The application uses Perlin [12] and Simplex [13] noise libraries to create realistic terrain. We chose these libraries for their ability to create noise using vectorized operations, notably enhancing generation speed. We will go into more detail about noise-based terrain generation in section 2.2

2.2 Algorithms and Data Structures

2.2.1 Backend Data Schema with Pydantic

Our application utilizes Pydantic for type safety, data validation, and efficient data exchange. Pydantic schemas define the data structures for algorithms, parameters, configurations, and user inputs, ensuring adherence to expected formats. The library also automates JSON serialization/deserialization for data exchange, minimizing code and error risks.

2.2.2 Noise Based Terrain Generators

In digital terrain generation, introducing noise can enhance realism by adding irregularities and natural-looking textures, similar to those found in the real world. Unlike the overly smooth surfaces seen in pure digital creations, natural landscapes have variations and imperfections. Noise algorithms like Perlin Noise and Simplex Noise mimic these natural irregularities, allowing for the creation of more lifelike models quickly and efficiently [14]. Perlin Noise is a popular method for generating natural-looking textures. It operates on a grid system, assigning gradient vectors instead of simple height values at each grid point. These vectors influence the texture by affecting the in-between areas, creating a more varied and realistic surface. This method involves less memory and is faster than many traditional techniques, though it becomes complex in higher dimensions. Simplex Noise, developed to address the shortcomings of Perlin Noise, excels in higher-dimensional spaces. By utilizing shapes with the minimum number of corners for each dimension, called simplices, it reduces the number of calculations needed. This makes it faster and more efficient, especially as dimensions increase. Although harder to understand and implement, Simplex Noise provides superior quality and is more suited for complex applications [14].

Perlin Noise

The provided Perlin noise algorithm generates realistic terrain data by applying a layered noise approach. This process involves iterating over the terrain multiple times, each with varying noise intensity and detail scale, referred to as octaves. Each octave modifies the terrain's appearance by adjusting the amplitude and frequency of the noise, making each layer progressively finer and less influential. This method ensures the generation of diverse terrain features, drawing inspiration from game terrain generation techniques described on "Red Blob Games" [15]. The algorithm employs the NPerlinNoise [12] library, enabling a full-grid noise application. This library simplifies generating complex noise patterns essential for realistic terrain modeling. Initially, the terrain data is either created from scratch or built upon existing data. The noise generation process then begins, scaling with the number of defined octaves. With each iteration, the persistence parameter reduces the amplitude, making subsequent layers subtler, while the lacunarity parameter increases the frequency, adding finer details. This iterative process accumulates diverse terrain features, layer by layer. Upon completing the octave iterations, the algorithm normalizes the resulting noise values, adjusting them to a 0 to 1 range. This normalization, combined with an exponent, controls the terrain's steepness. The output is a detailed terrain grid, expressed as a list of lists, ready for visualization. This method showcases how Perlin noise, through controlled repetition and variation, can produce complex and lifelike terrain models like shown in 2.2.

Simplex Noise

The Simplex noise algorithm, similar to the Perlin noise approach, is utilized for terrain generation. The Simplex Noise algorithm adheres to the same foundational principles, such as utilizing octaves, persistence, lacunarity, and normalization, but employs the Simplex method for a more refined output. And as demonstrated in Figure 2.3, fine-tuning the parameters can result in terrain that looks naturalistic. The Simplex algorithm progresses through a specified number of octaves, each contributing to the terrain's detail level. As with the Perlin method, the persistence reduces amplitude with each octave, while lacunarity increases the frequency, fine-tuning the terrain's details with each layer. The algorithm starts by either creating a new terrain grid or enhancing an existing one. Using the "opensimplex" library [13] it iteratively applies Simplex noise across the entire grid,

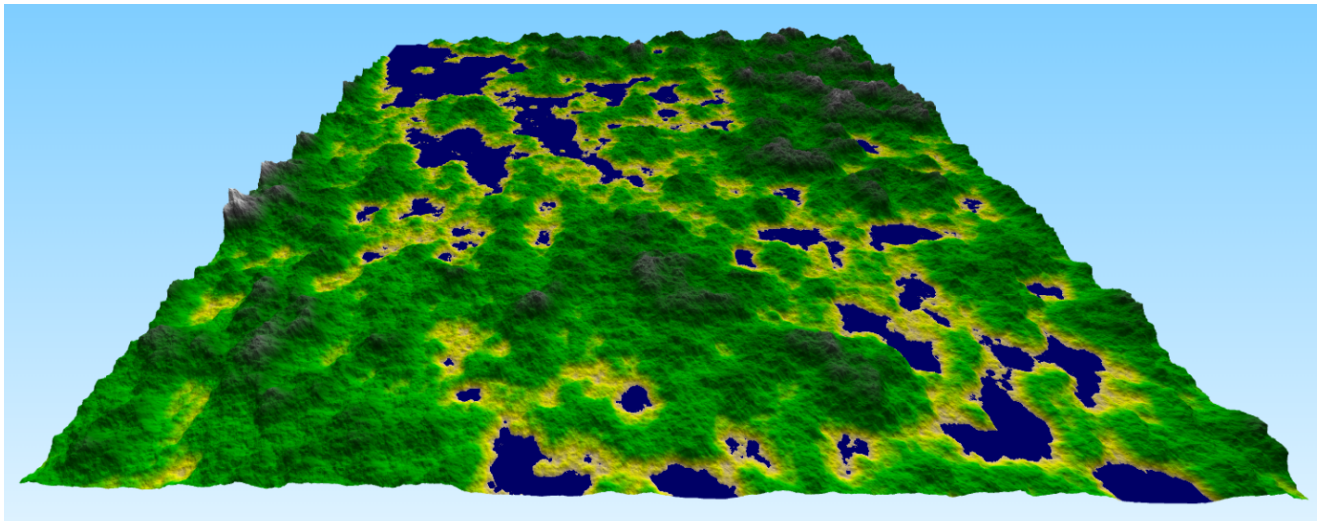


Figure 2.2: An example of a 1000x1000 terrain generated by the perlin noise algorithm.

adjusting amplitude and frequency according to the same principles as in the Perlin algorithm. Following the addition of each octave's influence, the terrain data undergoes normalization to ensure values fall within a 0 to 1 range, which is then adjusted for steepness using the exponent parameter. The end product is a grid, represented as a list of lists.

Noise Parameters

The parameters for both the Perlin and Simplex noise algorithms are crucial to adjust the generated terrain to specific needs. Here's a detailed look at each parameter:

- **Size**: Defines the resolution of the terrain grid. A larger size results in more detailed terrain, offering a finer representation of features. Smaller values produce less detail, ideal for broader landscape views.
- **Num_octaves**: Controls the number of detail layers. Higher octaves increase terrain complexity, adding intricate features. Fewer octaves result in a simpler, smoother landscape.
- **Persistence**: Affects amplitude reduction across octaves. High persistence yields a rugged terrain, as each layer strongly influences the landscape. Low persistence smooths out the terrain, diminishing finer details.
- **Lacunarity**: Dictates the frequency increase per octave. High lacunarity creates abrupt, detailed terrain changes, leading to a noisier look. Lower values offer a subtle detail increase, yielding a smoother landscape.
- **Max_height**: Scales the terrain's elevation range. A value of 1.0 uses the full height spectrum, from deep valleys to high peaks. Lowering this parameter compresses elevation differences, creating a flatter landscape.
- **Exponent**: Modifies terrain steepness. An increased exponent exaggerates elevation differences, accentuating peaks and valleys. This can make landscapes appear more dramatic and varied.
- **Seed**: Sets the random number generator's starting point. Different seeds produce unique noise patterns, enabling varied terrain formations without changing other parameters.

By adjusting these parameters, users can finely tune the generated terrain to match their needs.

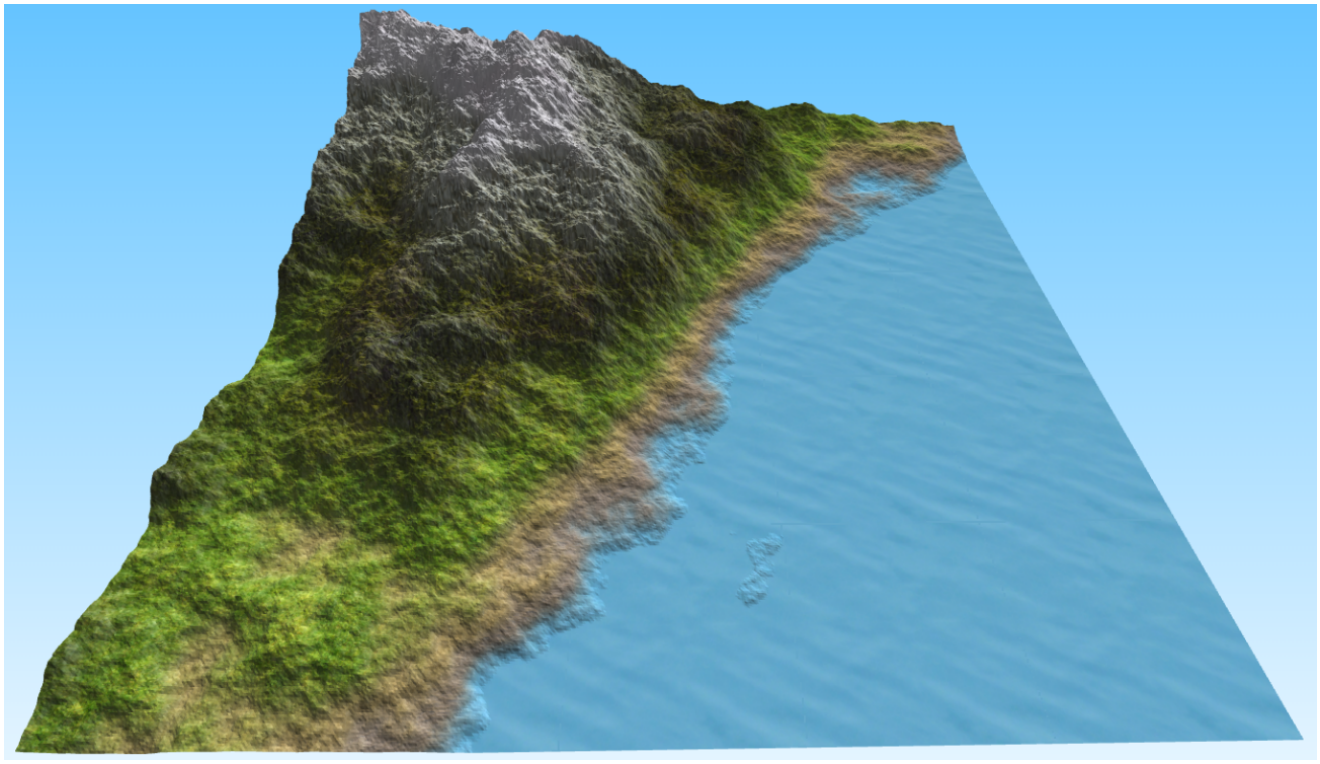


Figure 2.3: An example of a terrain generated by the simplex noise algorithm with applied textures.

2.2.3 Fractal Based Terrain Generators

Diamond Square Algorithm

The Diamond-Square algorithm is an enhancement of the Midpoint Displacement algorithm, addressing its directional artifacts by using four points for all calculations. It alternates between ”diamond” and ”square” steps to avoid linear artifacts and create more natural terrains. In the ”diamond” step, the algorithm averages the corners of a square and adds random displacement to form the center point. In the ”square” step, it averages the midpoints of a diamond’s sides, again adding random displacement. These steps reduce directional bias and create a more varied landscape [14].

If speed is crucial, the Diamond-Square Algorithm stands out because it’s usually faster than noise based methods [16]. However, edge cases pose challenges, particularly for pixels on the terrain’s boundary. Common solutions include averaging available points or wrapping points across edges for seamless tiling, though this can introduce other visual artifacts [14].

The underlying implementation of the Diamond-Square algorithm in the application initializes a terrain grid, then repeatedly applies diamond and square steps to fill in terrain details. The implementation provided is inspired on the approach described on [17]. One parameter controls the roughness, determining the displacement range, while the seed parameter sets the seed for the random number generator to ensure reproducability. As the visualization pipeline expects numbers between 0 and 1, the generated terrain values are normalized in the end.

The created Diamond-Square algorithm produces natural-looking terrain, which resembles real-world geographical features like showcased in Figure 2.4. Additionally, the algorithm’s outputs are great for stacking, allowing for the creation of multi-layered terrains and complex landscapes by overlaying multiple iterations of the algorithm with varying parameters.

L-System Terrain Generator

Incorporating the L-System algorithm into terrain generation offers a unique approach to creating patterns that emulate natural landscapes. The L-System, or Lindenmayer System, provides a method of fractal generation that

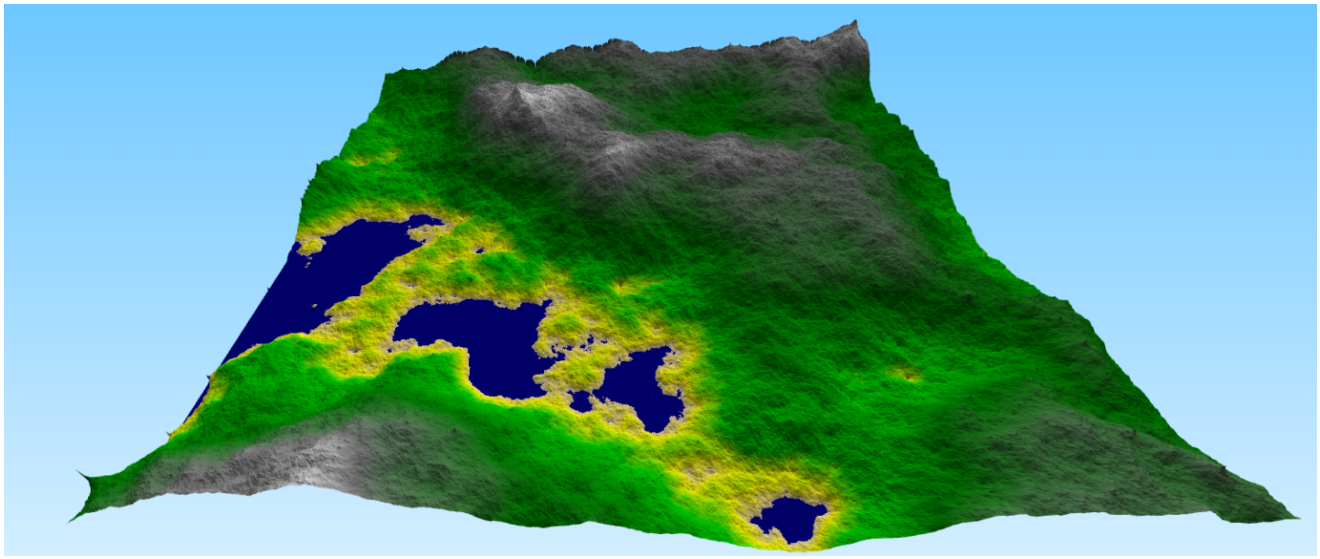


Figure 2.4: An example of a terrain generated by the diamond square algorithm.

can simulate the complex structures found in nature, such as tree branches, leaves, and even entire forest layouts. This generator leverages the principles of the L-System for the procedural generation of terrain. The L-System Terrain Generator is initialized with a set of parameters that define the characteristics of the generated terrain. These parameters include:

- **Size**: Determines the resolution of the terrain grid. A larger grid allows for more detailed and expansive terrains.
- **Axiom**: The initial string from which the L-System starts its generation process. This string is expanded through the application of production rules over several iterations.
- **Rules**: A dictionary defining the production rules of the L-System. Each rule specifies how a symbol in the axiom or a subsequent string is replaced with a new string of symbols, guiding the pattern's development. The L-System rules are interpreted as a string, with each symbol corresponding to a specific action:
  - **'F'**: Move forward and draw a line, contributing to the formation of terrain features.
  - **'+/-'**: Rotate clockwise or counterclockwise by the specified angle, altering the direction of terrain construction.
  - **'[' and ']'**: Push and pop the current state (position and orientation) onto a stack, facilitating branching and recursion within the terrain generation process.
- **Iterations**: The number of times the production rules are applied. More iterations result in a more complex and detailed pattern.
- **Angle**: Specifies the turning angle for directional changes, enabling the simulation of winding patterns like rivers or branching structures.
- **Direction**: Sets the initial direction of growth, influencing the orientation of the generated pattern.
- **Length**: Controls the step length for each move, affecting the scale of the terrain features.
- **Starting Point**: Determines where on the grid the generation process begins, which can centralize or offset the resulting pattern within the terrain.

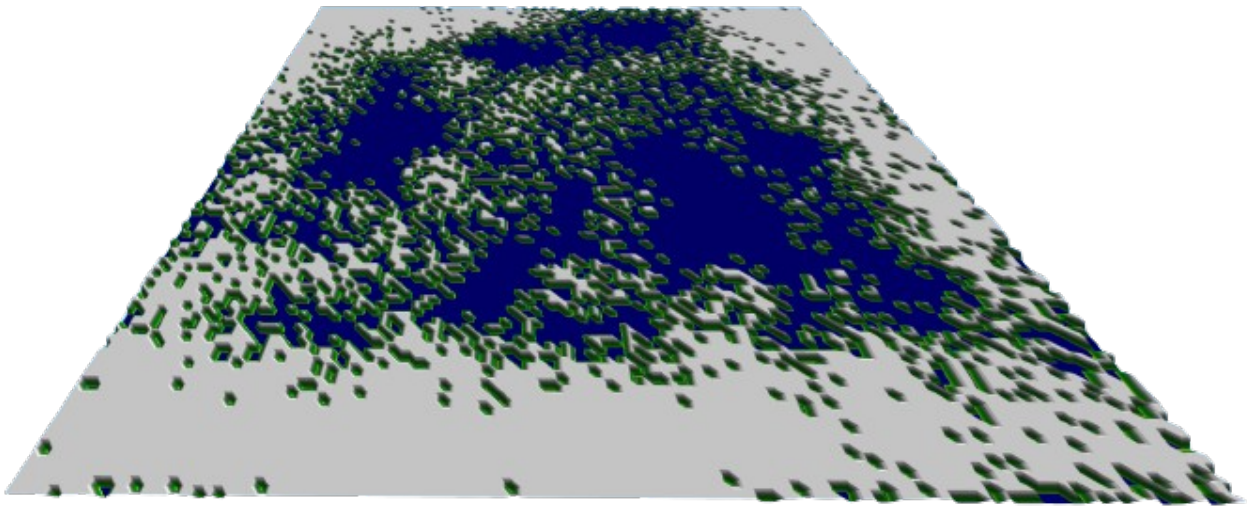


Figure 2.5: An example of a terrain created by the L-System algorithm.

The algorithm's core mechanism involves iteratively applying the specified production rules to expand the initial axiom into a longer string that directs the simulated growth process. This string is then interpreted as a set of drawing instructions, mapping the abstract pattern onto the terrain grid. Figure 2.5 showcases the creation of a L-System based terrain with a complex rule.

2.2.4 Voronoi Tessellation

One of our terrain generators leverages Voronoi tessellation to create terrain procedurally. The algorithm starts by generating a grid, the size of which is determined by the 'size' parameter. It then randomly places 'num_cells' points, known as cell centers, across this grid. These points will serve as the seeds for the Voronoi cells.

In Voronoi tessellation, space is divided into regions around each seed point. Each point in a region is closer to its corresponding seed point than to any other. This concept is central to creating our terrain, where each region represents a different terrain feature or elevation level [18].

In our algorithm, we iterate through each pixel in the grid. For every pixel, we find the closest cell center by measuring the Euclidean distance, a method highlighted for its natural occurrence and usefulness in grid generation [19]. The pixel's value is then set based on its distance to the nearest cell center, initially limited by a 'min_cell_radius' to ensure each cell has a minimum size.

However, the algorithm currently does not vary the height of each cell. Integrating height variation or other parameters could enhance realism, mimicking natural phenomena where terrain height does not uniformly change across regions.

After calculating distances for the entire grid, we normalize these values to a [0, 1] range, making them easier to interpret and visualize. The terrain height then can be adjusted by mapping these normalized values to a range of elevations using the UI.

In summary, the Voronoi Generator to divide space into regions based on proximity to a set of seed points. This method, rooted in both natural phenomena and mathematical principles, helps us generate terrain like patterns [18] [19]. Due to the flat nature of the surfaces generated, terrains created by the Voronoi algorithm may not appear very realistic without the addition of noise-based modifications as seen in Figure 2.6. However, if you wish to control the number of peaks within the terrain, a Voronoi-generated base can be really helpful. This can then be combined with either Perlin noise or Simplex noise to enhance the terrain's realism.

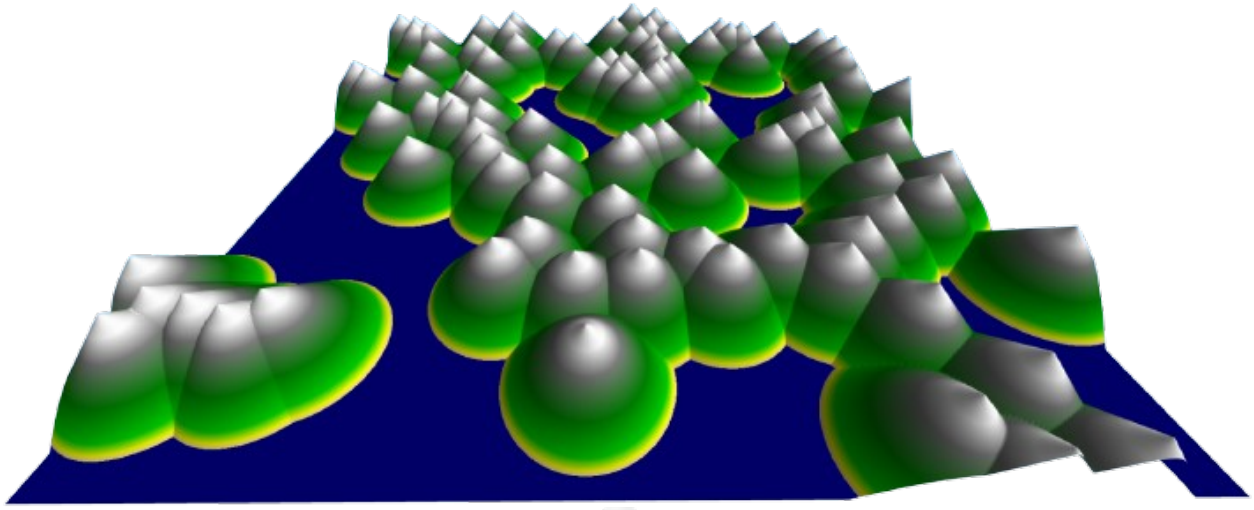


Figure 2.6: An example of a terrain created using Voronoi tessellation.

2.2.5 Hydraulic Erosion

Hydraulic erosion is the process of gradual decomposition by water, in our case the erosion of terrain by droplets of rain. Our hydraulic erosion method is largely based on the work by [20] and the open source C# implementation by [21], which we translated into Python. The implementation is a particle-based method where individual water droplets, simulated in 2D, move downhill, altering the terrain based on their capacity and speed. These droplets are tracked in two dimensions, do not interact with each other, and ignore physical principles. A unique aspect of this algorithm is that droplets move a fixed distance each step, independent of a grid, and the simulation time per step is not consistent. Therefore, this method is not designed for visual fluid simulation but to replicate the erosive effects of water on terrain, aiming to create visually appealing erosion patterns on both small and large-scale landscapes. An example of erosion visualized in our application can be seen in figure 2.7

Here is a brief explanation of each parameter. For more details and the formulas that use them, I recommend to read the extensive explanations in the work by [20].

- Seed: Seed for the random number generator. Different seeds result in different rain patterns.
- Erosion Radius: The radius of the droplet, or the radius from which a droplet captures sediment. Smaller values lead to thinner and more distinct ravines.
- Inertia: Controls the inertia of the droplet, meaning its ability to maintain a certain direction of movement after a change in slope. It must be between 0 and 1: Low inertia creates more pronounced valleys and ravines.
- Sediment Capacity Factor: Controls the sediment capacity of the droplet.
- Minimum Sediment Capacity: Controls the minimum sediment capacity of the droplet.
- Erosion Speed: Controls the erosion speed of the droplet. With higher speed, a drop quickly fills up its capacity and then most likely only deposits sediment. From 0 to 1.
- Deposition Speed: Controls the deposit speed of the droplet. Higher values lead to visible sediment deposition on the flow path. From 0 to 1.
- Evaporate Speed: Controls the evaporation speed of the droplet. A faster evaporation shortens the paths of the drops. From 0 to 1.

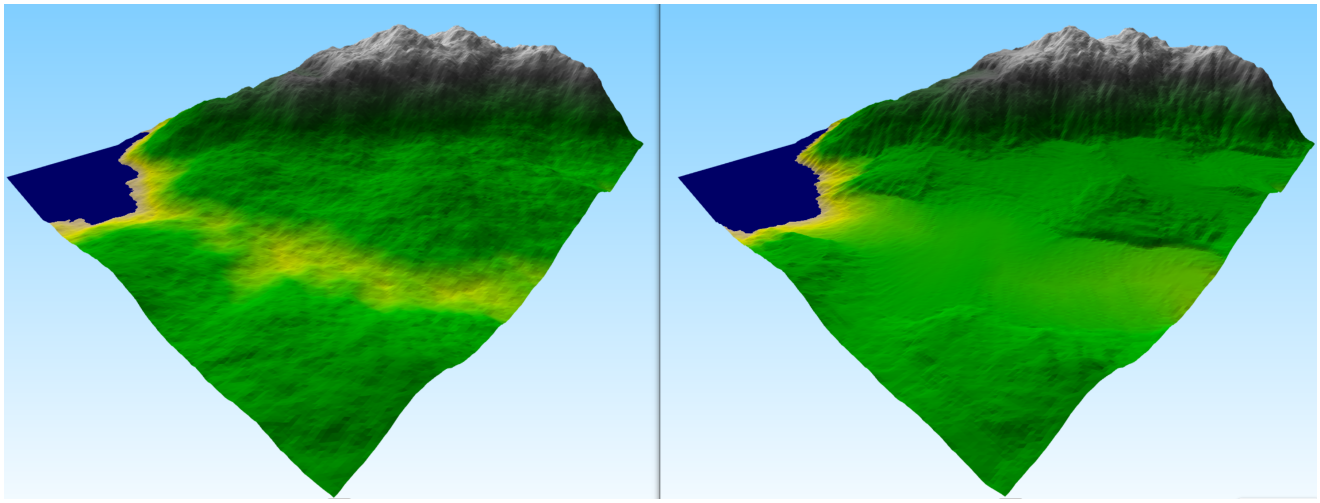


Figure 2.7: An example of hydraulic erosion applied on a 300x300 standard parameters Perlin noise terrain. The erosion has standard parameters, with 100'000 simulated drops.

- Gravity: Controls the gravity of the droplet, i.e. the speed of erosion, and therefore also the droplet capacity. No big impact on results.
- Maximum Droplet Lifetime: The maximum lifetime of the droplet, i.e. the maximum number of erosion steps it takes before it is deleted.
- Initial Water Volume: The initial water volume of the droplet.
- Initial Speed: The initial speed of the droplet.
- Number of Iterations: The number of iterations, or simulated water droplets, to perform.

2.2.6 Terrain Stacking

Both Perlin and Simplex noise techniques support the layering of different terrains. When generating terrains with one of these algorithms, there is an "Upload" button that enables users to upload a terrain in ASCII ESRI format. This feature allows for the integration of pre-existing terrains into the noise-generated landscapes. In the final step , both the newly created terrain and the uploaded terrain are first normalized, ensuring their values are within zero and one. They are then added together to combine their features. After this addition, the combined terrain is normalized once again. This ensures that both terrains have an equal influence on the final, stacked landscape. In Figure 2.8, an example is presented where Voronoi generated terrain is stacked on top of a Perlin Noise landscape.

2.2.7 Backend mesh and vertex normals generation

The initial solution that we found on a LearnWebGL forum thread [22] to generate vertex normals in the frontend turned out to be very inefficient: In fact, it took about 90% of the entire terrain processing time, probably caused by using a triple nested loop. Additionally, the old solution smooths the terrain by adding a midpoint for every square of four vertices (the squares in between coordinates), increasing the triangles count from two per square to four. We ultimately decided that it would be better not to use such a midpoint, because we wanted the user to more accurately see the actual generated data, and because this would allow generating and rendering larger terrains, effectively serving the purpose of smoothing that the midpoint solution offered.

We therefore set out to rewrite the mesh generation from scratch in the backend following the test-driven development methodology [1]. We later discovered that, especially with large terrains, the network latency of

[1] Tests can be found in `project_root/backend/testing/test_webgl_arrays_generation.py`

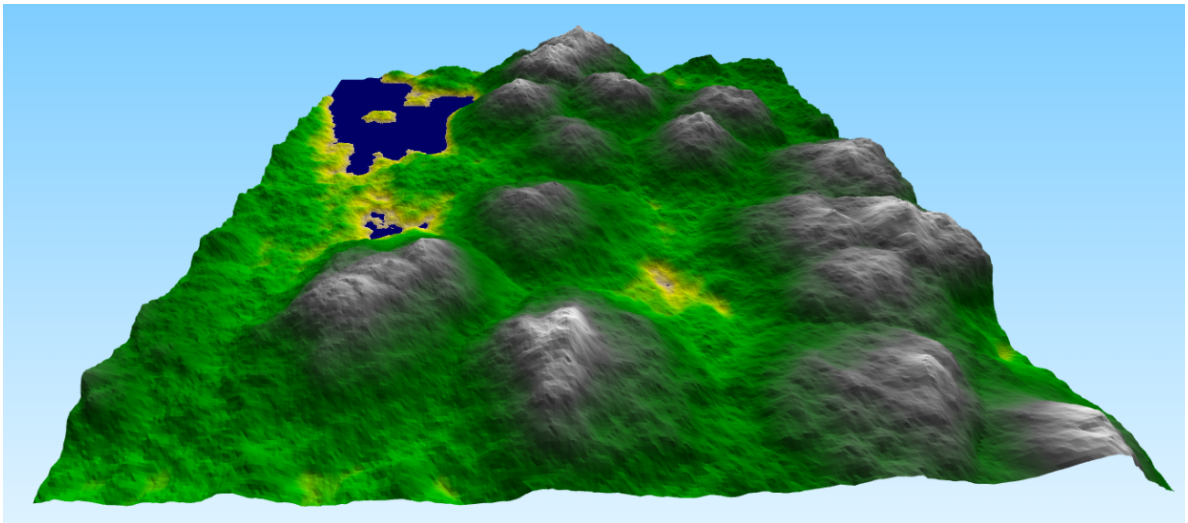


Figure 2.8: The result of stacking a Voronoi on top of Perlin Noise terrain.

sending the whole mesh to the frontend was too high. We thus decided to rewrite the mesh computation in the frontend based on our new code for it in the backend, and request only the vertex normals from the backend, which are generated very efficiently with NumPy's `gradient()` function.

These enhancements allowed to slash the terrain processing time from 33s to 5s for a 1024x1024 Perlin noise terrain, cached in the database (no generation). This is the time from clicking on "Generate terrain" to seeing the terrain, so it includes a little constant time for fetching the cached terrain from the backend and for the rest of the frontend pipeline.

3 Implementation

The source code related to the project is available on GitLab, as referenced in [23].

3.1 Source Code Organization

The project's code structure is divided into backend and frontend components. The backend's entry point is app.py, while index.js serves as the entry point for the frontend.

In the backend, we opted for a functional approach over an object-oriented one to maintain a simple and minimalistic API. However, there are exceptions to this rule, notably the Erosion class and the L-systems class. Given the complexity and multitude of variables involved in these components, incorporating a state-based approach was deemed more appropriate.

Similarly, in the frontend, we followed the same principle of utilizing React functional components whenever feasible. However, for modeling more complex objects such as the Renderer, we resorted to defining classes to manage complexity effectively.

3.2 Graphics Pipeline

Our final graphics pipeline spans across both backend and frontend. As explained in section 2.2.7, we rewrote the entire mesh and vertex normals computation code to improve speed. We currently compute the vertex normals in the backend in the API endpoint `/webgl_arrays` using the functions in `backend/webgl_helpers/webgl_arrays_generation.py`. We compute the mesh in the frontend in `frontend/src/visualization_classes/TerrainGeneratorNew.js`.

3.3 Frontend

Our frontend is structured with clear, modular components and services to ensure easy maintenance and navigation. The app's main parts are organized into components within corresponding folders. Custom hooks in the `hooks` directory simplify stateful logic and side effects, improving code reusability and readability. The `services` directory handles backend communication and data processing, keeping these operations separate from UI components.

Using Redux, the application's state is managed within the `store` directory, with reducers organized by feature for better modularity. Utility functions, such as form handling and terrain generation, are centralized within the `utils` directory. The root component connects various sub-components, manages global behaviors like theme toggling, and interfaces with Redux for state management.

Our frontend architecture is designed for extensibility, allowing for new features to be added with minimal reconfiguration. Integration with backend services is handled through a combination of Redux and service abstraction, ensuring a scalable and decoupled application design.

3.3.1 GUI

The Graphical User Interface (GUI) of our Parametrized Procedural Terrain Generation application serves as the primary point of interaction between the user and the terrain generation system. Designed with a focus on simplicity and usability, the interface facilitates easy navigation and operation for users of varying expertise levels.

The GUI is divided into several sections, each with a specific focus:

Control Panel

The Control Panel is a draggable floating panel that acts as the central hub for the terrain generation process. This panel features a segmented control system, enabling users to switch between 'Single', 'Comparison', or 'Multiple' generation modes. As users select a mode, the displayed interface adapts dynamically to match the chosen option. Each mode corresponds to a specific Form component, revealing algorithm-specific parameters that users can tweak to shape the terrain generation process (see Fig. 3.1).

History

The History Component keeps track of every terrain created, enabling users to revisit previous ones. It shows terrains along with their configurations, allowing users to download data or configurations for each entry. Additionally, users can compare different terrains directly from the history log. The interface is interactive, featuring keyboard navigation for terrain selection and a clear button to reset the history. (see Fig. 3.2).

Terrain Renderer

The Terrain Renderer consists of two main components: SingleRenderer and DoubleTerrainRenderer. The SingleRenderer displays a single terrain model on a canvas element. It relies on the handleRenderer utility function to update the displayed terrain whenever there are changes in the application state, such as the generation of a new terrain. In contrast, the DoubleTerrainRenderer activates when comparison mode is enabled. It presents two terrain models side-by-side, with an option to highlight the differences between them for a thorough visual comparison. (see Fig.

Comparison Selector

The Comparison Selector becomes active when comparison mode is enabled, facilitating a comparative analysis between two different terrains. It offers users an interface to choose from various comparison modes: 'Side by Side', 'Colored', 'Main Terrain Only', or 'Second Terrain Only'. A designated 'Exit Comparison' button allows users to easily revert to viewing single terrain models after comparison.

Height Calculator

The Height Calculator tool uses the terrain size to determine the highest point on the terrain based on user-input parameters like side length, exponent, and height multiplier. It uses the application state to compute the height in meters, aiding users in understanding the terrain's features.

Loader

The Loader component serves as an overlay featuring a loading animation and a brief message to users during operations involving wait times, like terrain generation or data fetching.

State Management Behind the scenes, state management is handled by Redux, ensuring that the user's interactions are smoothly translated into visual changes within the application. This robust management system maintains application state across user sessions and interactions, providing a seamless experience.

The GUI is tightly integrated with the Visualization Classes, directly controlling the rendering of terrains based on user input. When users interact with the GUI, the Visualization Classes receive updated parameters and regenerate the terrain accordingly, providing a seamless flow from user input to visual output.

3.3.2 Visualization Classes

Renderer

The Renderer class handles terrain visualization in a WebGL context, leveraging the ShaderProgram class for WebGL setup and shader management. The main purpose of this class is to manage the rendering loop, handling

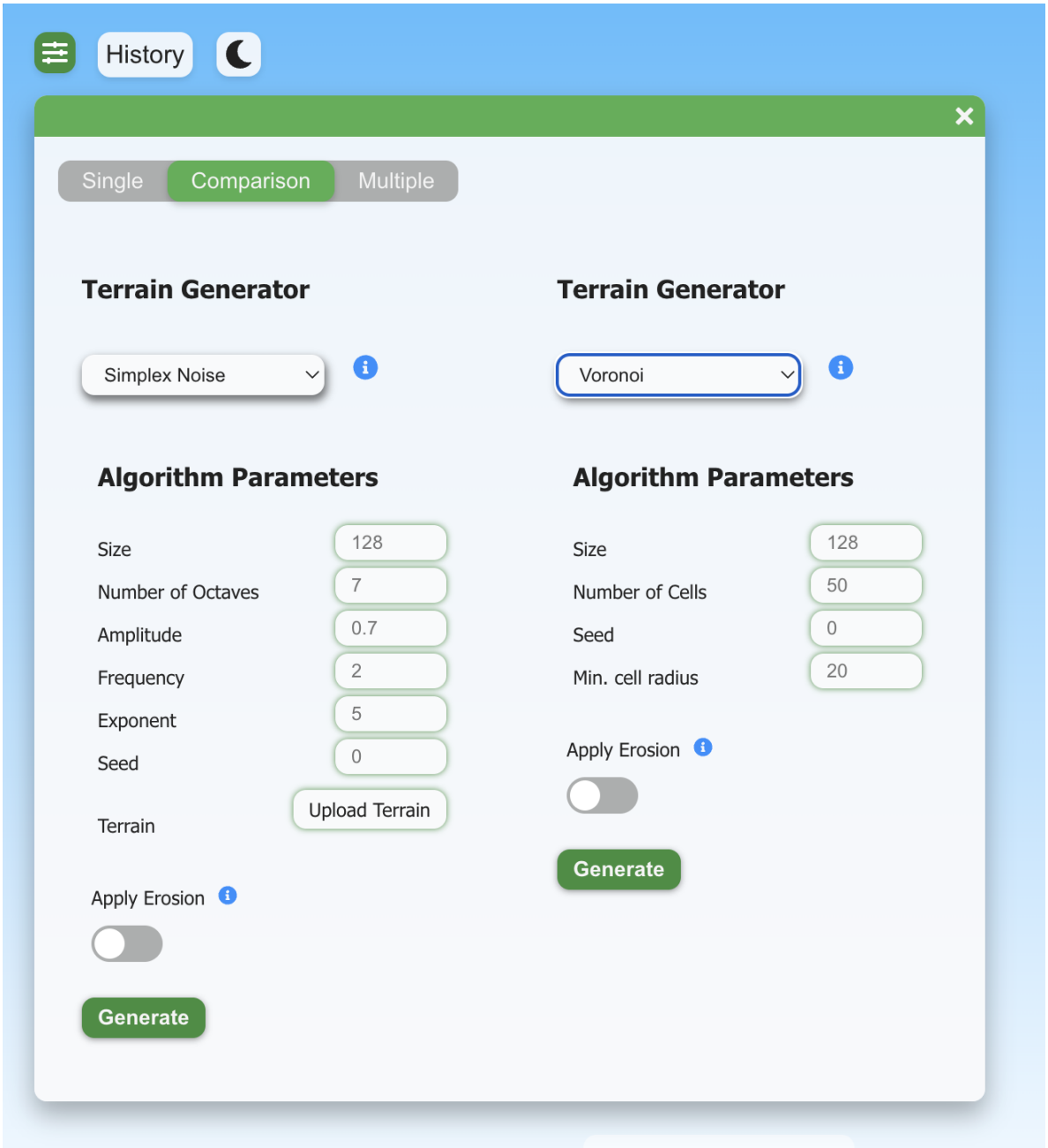


Figure 3.1: Control Panel used for generating two terrains.

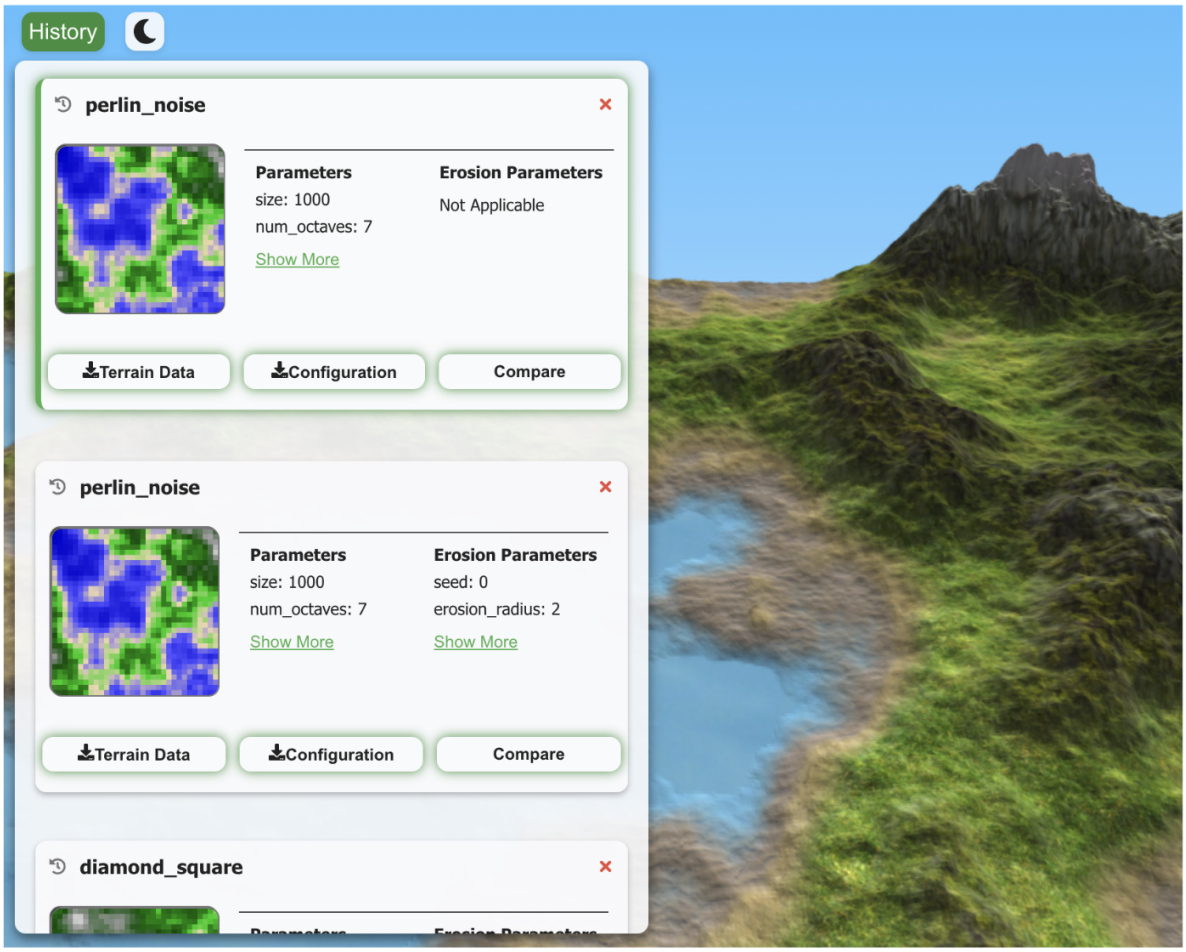


Figure 3.2: History Panel keeping track of previously generated terrains

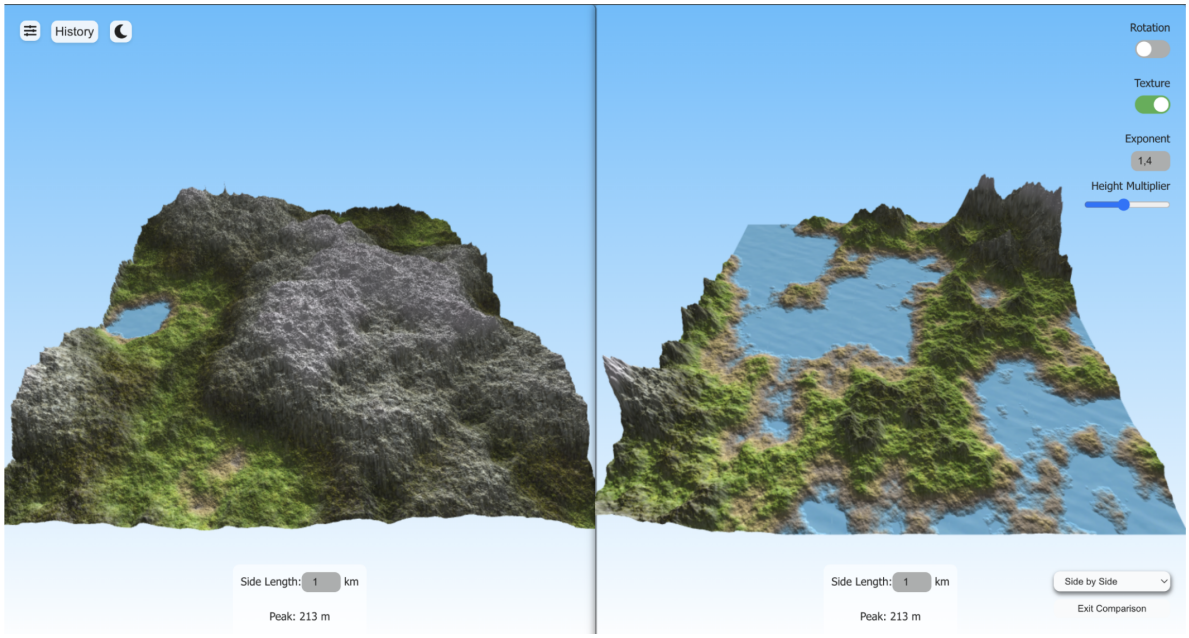


Figure 3.3: Terrain Renderer component displaying two generated terrain side by side

matrix operations for translation, rotation, and scaling, based on user inputs and predefined settings. Other features of this class include dynamic terrain rendering, UI controls for terrain customization, and an efficient terrain data update system. This allows the terrain to be updated without the need to create a new Renderer object, ensuring good performance. Furthermore, it interacts with the ShaderProgram object to apply visual effects and ensure that user-selected textures reflect in the rendered terrain.

In summary, the Renderer class acts as the central hub for the visualizations of terrains, handling everything from graphics settings and user inputs to the final rendering process.

Shader Program

The ShaderProgram class is essential for managing WebGL shader programs. It handles the setup of WebGL contexts, compiles shaders, and links them into a program. The class is designed for terrain visualization, utilizing vertex and fragment shaders to render the created mesh. Key functionalities include establishing a WebGL2 context, loading and compiling shaders, creating and managing textures, and setting shader parameters dynamically.

Once instantiated with a canvas selector by the Renderer Class, the ShaderProgram must be initialized to prepare for rendering. This involves loading shaders, compiling them, and setting up textures and uniform parameters. The Shader Program Object then interacts with the GUI trough the Renderer Class in real-time, allowing users to select and display textures, thus dynamically affecting the visual output. As the name suggests, it forms the backbone of the shader program, visualising content based on the supplied shaders and parameters.

3.3.3 TerrainGeneratorNew and TerrainGeneratorMidpoint

`TerrainGeneratorNew` is the new class for computing the mesh of our generated terrains, as described earlier in section 2.2.7. It replaces `TerrainGeneratorMidpoint`, which was based on a less efficient solution found on StackOverflow [22], also described in section 2.2.7.

3.3.4 Graphics

Vertex Shader

The main functionality of the vertex shader revolves around modifying the terrain's geometry based on user-controlled parameters. It adjusts vertex heights using a 'heightMultiplier' and an 'exponent', which users can change via the interface. This stretching process is important for applying the height transformation.

Firstly, the shader calculates the new y-coordinate (height) of each vertex by multiplying the original height by the 'heightMultiplier' and then raising the result to the power of 'multiplierExponent'.

Secondly, the modified position ('stretchedPosition') is used to compute the final position of the vertex in the scene ('gl_Position'), applying the model-view-projection matrix ('mvp').

Additionally, the shader calculates normals for lighting ('v_normal'), handles texture coordinates ('v_texcoord'), and determines the view direction ('v_viewDirection'). Furthermore, it computes the fragment's position in world space ('v_fragPosition'), necessary for further calculations in the fragment shader.

Fragment Shader

The fragment shader is responsible for coloring and shading each pixel of the terrain. It uses predefined colors and offsets in the default mode to apply color based on the terrain's height. For example, lower heights might be colored as water, while higher ones could resemble forest or snow. This process ensures that the terrain is rendered with realistic color gradients and transitions.

Texturing

The user can also apply textures to the terrain. These preloaded textures are applied in a repeating grid pattern. The specific texture displayed varies based on the height of the terrain, allowing different areas to be visually distinguished, similarly to the default mode. Using the smoothstep function, the shader creates soft transitions

Endpoint	Methods	Input parameters	Description
`/available_algorithms`	GET	None	Returns JSON with parameters and descriptions for each algorithm
`/erosion_parameters`	GET	None	Returns JSON with parameters and descriptions for the erosion algorithm
`/generate_terrain`	POST	`selectedAlgorithm: str, terrainParameters: dict, hashedParams: str, erosionParameters: dict, applyErosion: bool`	Generate, erode, and store terrain data based on the specified algorithm and the terrain and erosion parameters.
`/downsample_terrain`	GET	`hashedParams: str, targetSize: int`	Get a downsampled version of a stored terrain.
`/store_terrain`	POST	`hashedParams: str, terrainData: list[list[float]]`	Store a terrain by its hashed parameters as key in a document store. (No longer used internally)
`/retrieve_terrain`	GET	`hashedParams`	Retrieve a stored terrain.
`/webgl_arrays`	POST	`hashedParams, whichArraysFromBackend`	Generate any combination of: Triangle indices, vertex positions, vertex normals, and texture coordinates arrays of a terrain stored in cache for WebGL2.

Figure 3.4: The API of the project.

between different textures, enhancing realism. When the terrain is steep, as indicated by the normals, the texture smoothly transitions to a ”stone” texture to simulate rocky surfaces.

Lightning

Lighting effects are applied to the textured terrain within the fragment shader. The shader calculates ambient, diffuse, and specular lighting based on the terrain’s material properties and the light’s characteristics, such as direction and intensity. This approach adds depth and realism to the scene, with brighter areas indicating direct light exposure and darker areas representing shadows.

3.4 Backend

3.4.1 app.py, API

The API defined in `app.py` allows users to use the project programmatically. It is written following a functional paradigm using Flask. The API is described in figure 3.4. Figure 3.5 shows a sequence diagram of an example workflow showcasing the use of the API with both the frontend’s GUI and programmatically, as well as the relationships between backend, frontend, and mongoDB.

3.4.2 Erosion

The hydraulic erosion described in section 2.2.5 is written as a Python class. This differs from the usual functional code of the backend, but it was a necessary decision due to the complexity of the code, which includes many attributes used by several methods.

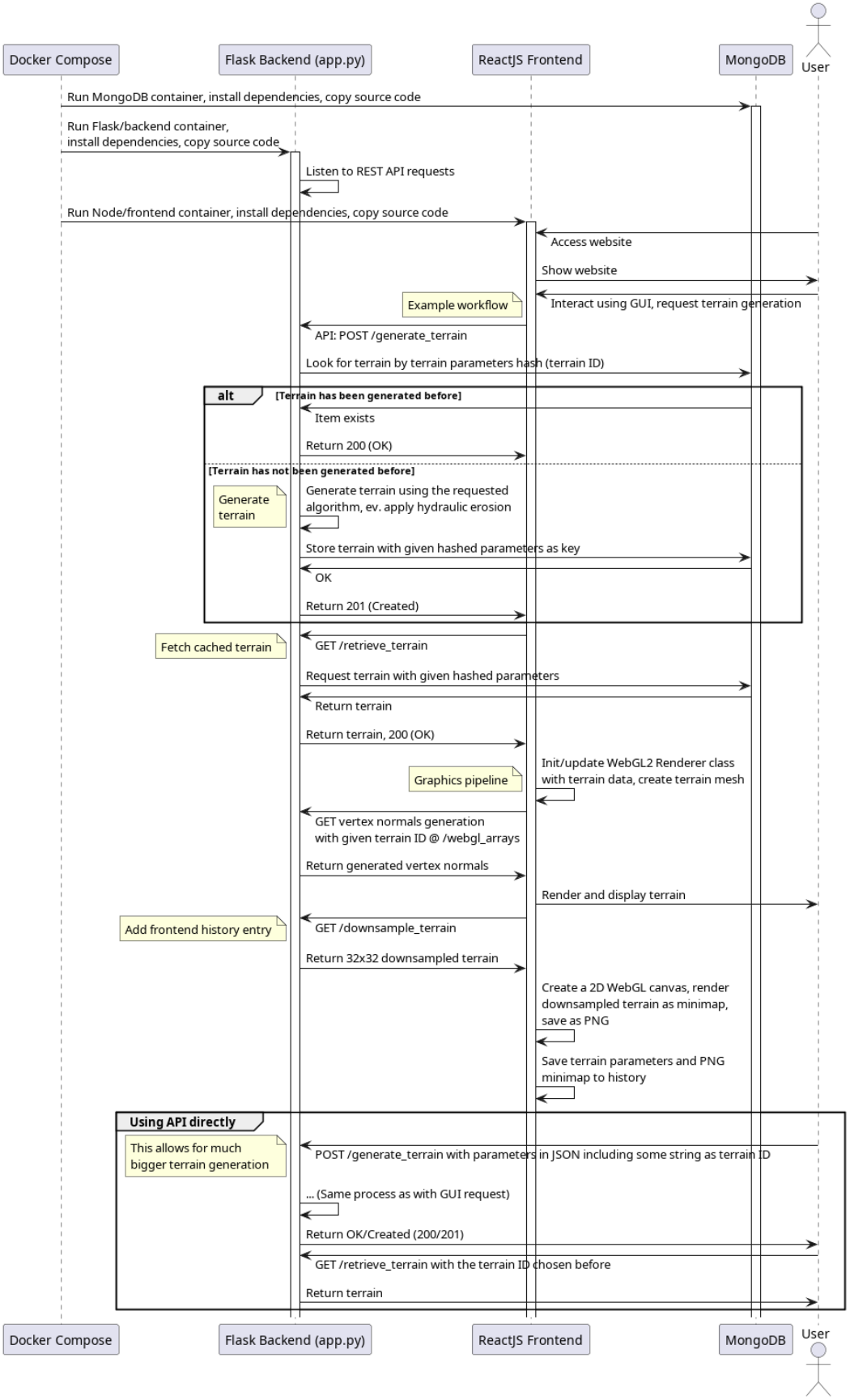


Figure 3.5: Sequence diagram of an example workflow showcasing the use of the API with both the frontend's GUI and programmatically

4 Conclusion and Discussion

4.1 Conclusion

In conclusion, this project has successfully established a comprehensive framework for parametrized procedural terrain generation, offering an innovative and interactive tool for the creation of varied terrains. Through the integration of advanced algorithms and a user-friendly interface, we have demonstrated the potential for significantly enhancing the quality and realism of generated terrains. Our achievements not only highlight the technical feasibility and efficiency of our approach but also pave the way for further research and application in diverse fields requiring realistic terrain simulations. The project stands as a testament to the potential of combining procedural generation techniques with intuitive user controls, setting a new benchmark for future developments in terrain generation technology.

4.2 Future work

While our project has accomplished its objectives, there are areas where further improvement is possible. One such area is the implementation of an erosion history feature. This feature would enhance the visualization by allowing the color mapping of rivers and lakes based on their erosion patterns.

Another potential enhancement involves optimizing terrain comparison speed. One approach could be to implement a cache system within the Redux state to store frequently accessed terrains temporarily, thereby reducing loading times. Additionally, caching vertex normals in MongoDB along with terrain data could expedite data retrieval processes.

Lastly, some remaining minor bugs need to be addressed. These issues, which either surfaced during the final stages of development or have persisted despite previous efforts, require attention to ensure the overall stability and functionality of the project.

Acknowledgements

- Starting code to work with height maps in WebGL:
  https://webglfundamentals.org/webgl/lessons/webgl-qna-how-to-import-a-heightmap-in-webgl.html
- Illustrative explanations for creating terrain using noise functions:
  https://www.redblobgames.com/maps/terrain-from-noise/
- Base code for mesh generation in Python:
  https://loady.one/blog/terrain_mesh.html
- Hydraulic erosion C# code by youtuber Sebastian Lague, translated by us into Python:
  https://github.com/SebLague/Hydraulic-Erosion
- Diamond square algorithm based on:
  https://learn.64bitdragon.com/articles/computer-science/procedural-generation/the-diamond-square-algorithm

Bibliography

[1] T. Hyttinen, E. Mäkinen, and T. Poranen, "Terrain synthesis using noise by examples," in *Proceedings of the 21st International Academic Mindtrek Conference*. Tampere Finland: ACM, Sep. 2017, pp. 17–25. [Online]. Available: https://dl.acm.org/doi/10.1145/3131085.3131099

[2] Docker, "Docker: Accelerated Container Application Development," May 2022. [Online]. Available: https://www.docker.com/

[3] A. Ronacher, "Flask: A micro web framework written in Python." [Online]. Available: https://flask.palletsprojects.com/en/3.0.x/

[4] J. Alakuijala, A. Farruggia, P. Ferragina, E. Kliuchnikov, R. Obryk, Z. Szabadka, and L. Vandevenne, "Brotli: A General-Purpose Data Compressor," *ACM Transactions on Information Systems*, vol. 37, no. 1, pp. 4:1–4:30, Dec. 2018. [Online]. Available: https://dl.acm.org/doi/10.1145/3231935

[5] M. Inc., "MongoDB: The Developer Data Platform." [Online]. Available: https://www.mongodb.com

[6] "google/snappy," Mar. 2024, original-date: 2014-03-03T21:58:09Z. [Online]. Available: https://github.com/google/snappy

[7] "GridFS," in *The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing*, E. Plugge, P. Membrey, and T. Hawkins, Eds. Berkeley, CA: Apress, 2010, pp. 83–95. [Online]. Available: https://doi.org/10.1007/978-1-4302-3052-6_5

[8] Facebook, "React: A JavaScript library for building user interfaces." [Online]. Available: https://react.dev/

[9] Redux, "Redux - A predictable state container for JavaScript apps. | Redux." [Online]. Available: https://redux.js.org/

[10] K. Group, "WebGL 2.0." [Online]. Available: https://registry.khronos.org/webgl/specs/latest/2.0/

[11] Greggman, "TWGL.js, a tiny WebGL helper library." [Online]. Available: http://twgljs.org

[12] A. M, "amithm3/nPerlinNoise," Feb. 2024, original-date: 2021-01-24T10:23:21Z. [Online]. Available: https://github.com/amithm3/nPerlinNoise

[13] Alex, "lmas/opensimplex," Mar. 2024, original-date: 2015-04-13T15:11:53Z. [Online]. Available: https://github.com/lmas/opensimplex

[14] T. Archer, "Procedurally generating terrain," in *44th annual midwest instruction and computing symposium, Duluth*, 2011, pp. 378–393.

[15] A. J. Patel, "Making maps with noise," Red Blob Games, Tech. Rep., 2015. [Online]. Available: https://www.redblobgames.com/maps/terrain-from-noise/

[16] T. J. Rose and A. G. Bakaoukas, "Algorithms and Approaches for Procedural Terrain Generation - A Brief Review of Current Techniques," in *2016 8th International Conference on Games and Virtual Worlds for Serious Applications (VS-GAMES)*. Barcelona, Spain: IEEE, Sep. 2016, pp. 1–2. [Online]. Available: http://ieeexplore.ieee.org/document/7590336/

[17] "The Diamond Square Algorithm." [Online]. Available: https://learn.64bitdragon.com/articles/computer-science/procedural-generation/the-diamond-square-algorithm

[18] K. S. Emmanuel, C. Mathuram, A. R. Priyadarshi, R. A. George, and J. Anitha, "A Beginners Guide to Procedural Terrain Modelling Techniques," in *2019 2nd International Conference on Signal Processing and Communication (ICSPC)*, Mar. 2019, pp. 212–217.

[19] L. Ju, T. Ringler, and M. Gunzburger, "Voronoi Tessellations and Their Application to Climate and Global Modeling," in *Numerical Techniques for Global Atmospheric Models*, P. Lauritzen, C. Jablonowski, M. Taylor, and R. Nair, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, vol. 80, pp. 313–342, series Title: Lecture Notes in Computational Science and Engineering. [Online]. Available: https://link.springer.com/10.1007/978-3-642-11640-7_10

[20] H. T. Beyer, "Implementation of a method for hydraulic erosion," Nov. 2015.

[21] S. Lague, "Hydraulic-Erosion," Aug. 2023, original-date: 2019-02-21T11:32:29Z. [Online]. Available: https://github.com/SebLague/Hydraulic-Erosion

[22] Cosmo and Gman, "How to import a heightmap in WebGL," Dec. 2019. [Online]. Available: https://webglfundamentals.org/webgl/lessons/webgl-qna-how-to-import-a-heightmap-in-webgl.html

[23] C. Skorski, P. Kalliopi, and K. Moser, "vmml / terrainGeneration · GitLab," Jul. 2023. [Online]. Available: https://gitlab.ifi.uzh.ch/vmml/terraingeneration