Bacheloroppgave

NTNU
Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for datateknologi og informatikk

Robin Sandvik, Viktor Chambe-Eng

Flexible procedural terrain and biome generation

Bacheloroppgave i Bachelor i programmering
Veileder: Mariusz Nowostawski
Mai 2023

NTNU
Kunnskap for en bedre verden

Robin Sandvik, Viktor Chambe-Eng

Flexible procedural terrain and biome generation

Bacheloroppgave i Bachelor i programmering
Veileder: Mariusz Nowostawski
Mai 2023

Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for datateknologi og informatikk

NTNU

Kunnskap for en bedre verden

Abstract

This thesis describes a highly flexible procedural generation framework that can be integrated with other programs, such as games, modelling and visualization software. The product is made in Unity3D (a popular 3D game engine) and it can be used for Unity3D and other projects that do not use Unity3D. Our procedural generation framework can be used to develop maps and terrain, whether for games or visualizations of 3D worlds.

The program breaks the process of generating terrain down into several individual steps and provides a collection of generators that handle different steps in the process (referred to here as step generators). These steps include processes like determining the altitudes of the landscape, spawning rivers, and simulating wind. Users can pick and mix different step generators from a wide selection of pre-made algorithms. These steps all have parameters that allow the user to customize how the terrain is constructed, such as how rough the altitudes should be, how many rivers should be included, or how sensitive the wind should be to altitude changes. The users can also easily make their own step generators with minimal extra code required, to make them fit their specific needs. The toolbox is designed to be effective when testing and developing a game, when designing and curating maps, and when procedurally generating at runtime.

The main motivation for our project came when Progress Interactive AS, a small indie game studio, approached us wanting a procedural terrain generation program they could use to aid in making maps for a real-time strategy game they have been working on. Our solution fits their game requirements. In addition, the product is also made to be highly flexible and easy to use in any other use case that requires procedural terrain generation.

Acknowledgements

We want to thank our advisor Mariusz Nowostawski, who gave us much valuable advice and feedback.

Table of contents

Glossary

Procedural	Follows a procedure (Sequence of steps). Procedural generation differs from random generation in that it is meant to produce the same outcome if given the same input.
Indie	Independent, generally used for game developers that do not have the backing of big companies.
JSON	JavaScript Object Notation. A popular format for storing data in text format.
Open Source	A publication license for code where the original source code is freely available for anyone to read. Open source code is often free to use however you like without payment, but open source code does not have to be free to use.
GUI	Graphical User Interface.
RGBA	Red Green Blue Alpha. A common format for storing image data where each pixel is stored in 4 values representing how much red, green and blue the pixel has, with the 4th value, alpha, representing transparency.
Library	In software, a library is a collection of code someone has made that others can use in their own code instead of having to make it themselves.
Class	In programming, a class is a type of object. The class of an object describes what information it stores and what it can do.
RTS	Real-Time Strategy. A genre of strategy games where everything happens in real time, as opposed to each player getting a turn.
Users	In this report, we will use the work **user** to refer to someone using our program. This could be anyone, from developers using it in their own code, to people playing a game (referred to in this report as **players**) and using our program indirectly.

1 Introduction

Stakeholders

Progress Interactive AS is an independent third-party contractor for game development services. It provides 2D and 3D artists, and programmers to larger game studios and projects. Currently, they are in the early stages of developing a new real-time strategy game. The company has been struggling with making affordable games in a market dominated by bigger companies. This problem has been made even worse by the recent pandemic. To that end, they have tasked us with finding practical solutions for cutting down the cost and time to make games without having a significant negative impact on the product. Specifically, they want us to develop a tool that can procedurally generate terrain for their new real-time strategy game. This terrain should have altitudes, biomes, and resources.

Team background and competences

- Both team members had Bachelor in Programming as their degree.
- Both team members had some experience with Unity from the Game Programming class taken the semester before. One team member had a decent bit of experience with C# in general.
- The team had some familiarity with algorithms and advanced math in general.
- Neither team member had much experience with procedural generation or terrain making.
- Both team members have played RTSes, giving insight into how maps for these games are usually structured.

Other roles

- Richard Barlow was our representative from Progress Interactive AS.
- Mariusz Nowostawski was our supervisor from NTNU.

Problem area

Developing a new strategy game, especially a resource focused one, is very challenging. The game needs to preserve a good balance to keep it fair as well as competitive, and with every new feature or change, the balance shifts greatly. To try out new mechanics and ideas to see if they work, the developers need maps to test it on, but creating a new map requires time and effort. Additionally, as changes are done to the game's mechanics and assets, old maps may no longer be compatible. This means that new maps have to be made as they develop the game. This adds a lot more work and slows development down greatly.

In addition to testing, the game needs to have actual playable maps for the players to use. These can either be pre-made and curated by the developers, or randomly generated on the player's computer at runtime. In both of these cases, procedural generation can be used. It is important that the generated maps are varied and interesting, while retaining certain attributes and patterns that the developers want.

For this reason, they have requested us to create a program that allows them to procedurally generate maps they can use for the game, both as a tool for the developers and to be integrated into the game. This tool will be made open source, and is also designed to be easy to use, expand, or integrate into any game that requires procedural terrain generation.

Boundaries

- Our program was made with C# in Unity, but the design can be recreated in other languages relatively easily.
- Our program only focuses on generating maps. It will not focus on implementing a game with the maps. The program is meant to be used either to generate maps, then load them into a game, or used in the game itself to generate the maps.
- In this project we were not focused on parallelization or optimization, though we did keep the performance reasonable.

Task definition

For this project we were asked to create a program that Progress Interactive could use to generate maps for their games.
They intend to use this in 2 main ways:

1) To generate maps for testing as they develop the game.
2) To make a set of maps to release with the game.

They were also considering using it to let players generate new maps when starting a game. They eventually decided this was not a high priority, but that keeping that as an option would still be useful.

The maps consist of rectangular tiles that represent an area of land. Each tile should have an altitude, resources, and belong to a biome. There should be different types of biomes that look and behave differently, such as forests, deserts, jungles and rivers. Biomes should also follow certain logic, such as rivers and lakes providing water to nearby tiles, affecting what kinds of biomes can spawn. The system should also be able to convert the maps into files that can be loaded elsewhere. This is to allow developers to save and load maps, and to let maps be shared around.

In addition to what the employer requires, we wanted to make the program flexible enough to be used in other games. We want to make the project open source, and useable by both developers who want a quick and simple tool and those who want to customize and expand on our code.

Target audience

This report is intended for readers with some programming knowledge, though it need not be specific to C# or Unity. While this report focuses mainly on the program we made, it contains information relevant to procedural generation and framework design in general. It can therefore also be used to get information and inspiration for these topics.

The product is intended for game developers who want a tool that can generate a wide variety of tile-based maps. This applies both to developers who want a quick solution, and those who want to customize and expand our code to tailor the maps to their specific needs. While it is made in Unity, it can be replicated in another language without too much work. The maps can be generated in Unity, then loaded into another program. Alternatively, the program can be integrated directly into a game.

Frames

- The project had to be completed by the 22nd of May.
- The program had to be possible to use in Unity.
- We had to use a BitBucket repository provided to us by our employer.

Report structure

Chapter	Purpose
1 Introduction	Gives a brief introduction to the thesis and the context around it.
2 Background	Breaks the task down and goes into theories and principles relevant to the project.
3 Requirements	Builds a set of requirements the solution should fulfil based on the assignment and theory.
4 Design	Goes into the overall design of our solution.
5 Development process	Describes our process for working on this thesis.
6 Implementation	Goes more in detail about how different parts of the solution are implemented.
7 Testing	Describes how we tested our solution and made sure we kept on the right track.
8 Deployment and integration	Describes how to deploy and use our program.
9 Summary	Summary of the project and conclusion to the thesis.

2 Background

Procedural generation design philosophy

To create the ideal procedural terrain generator for a given scenario, there are a lot of things to consider.

Procedural Generation metrics and quality assessment

Firstly, the generated maps must be good. What counts as a good map is highly subjective and depends on what it is used for. Some factors that may be desirable in some games may be in direct opposition to ones that are desirable in others. Here we have outlined the most important factors:

- **Balanced**: If one player has a significant advantage over another, the game becomes unfair.
- **Strategic**: Areas should have different benefits and drawbacks to encourage prioritizing certain locations over others.

- **Informative**: By looking at an area, a player should be able to draw conclusions about the surrounding area.
- **Unpredictable**: There should be room for surprise and a need to explore.

- **Interesting**: If everything looks the same, the map becomes boring.
- **Consistent**: Things should not look too chaotic and random.

- **Realistic**: Areas should imitate how nature behaves in reality.
- **Stylized**: Areas should look aesthetically pleasing and follow a consistent art-style.

Since the game is in the early stages of development, and the game mechanics and aesthetics are in constant flux, it is difficult to know which of these factors will be most important. Exactly what determines how well a factor has been achieved may also change. For instance, if a new game mechanic is added, a previously well balance map may become unbalanced. Because of this, it is difficult to make a specialized generator this early in development.

Uses of procedural terrain generation

Secondly, the way the program is used has a big impact on what assumptions we can make and the level of quality we need to provide.

The program could be used to:

- Test mechanics while developing the game.
- Create curated maps the developer wants to save and use in the finished game.
- Have the game generate complete maps on the go, such as for a new match.

When the user is generating maps to test on while developing the game, the quality doesn't need to be perfect. Most of the time, the developers are testing a particular change, and more focus will be on

getting results fast so they can make changes and try again. It is important that the generator is able to adapt to their changes. If they add a new resource, they need to be able to generate it on the map without too much work. If it takes as much time updating the generator as it would making the map manually, then it does not help with their problem.

When making maps to release with the game, quality and control has a lot more importance. These maps should ideally be curated by the developers to make good use of their game's mechanics and give a good impression of the game. This means that we need to allow for manual changes of the map and allow users to quickly iterate, making sure they receive immediate feedback.

Lastly, when generating new maps on the go, the priorities are completely different. In this case, the map generates without any user feedback. The map we generate is the map they get, so the quality becomes important, since we must get everything right on the first try. Another important factor here is customizability. We need to provide the user with a way to inform us about how they want their map before the generation starts. This could be anything from the map size, to the number of players, to resource density and general climate.

Chunk-based procedural generation

Another important question is whether the whole map should be created in a single pass, or if it should be able to create it one chunk at a time. Being able to create the world one chunk at a time has many benefits, such as being able to expand the map in real time, thus resulting in potentially infinite content on each map. It also significantly reduces the memory cost at runtime, as the map only has to be loaded as far as the user can see, rather than all at once. However, it adds a lot of restrictions on what we can do. If new chunks can be generated as we go, we can't generate based on surroundings, since they are not always known.

For this project, we decided to prioritize generating the whole map at the start, as we found the benefits of chunk-based generation were outweighed by the drawbacks, and the employer agreed. If another user requires chunk-based generation, it is still possible to make changes to our program and add that support. However, since most of the algorithms we implemented are not designed to work with only part of the map, new algorithms would need to be added.

Benefits of a flexible framework

In short, our generator needs to:

- Be "good" for the game, player and playstyle it is used in.
- Be able to make a good map on its first try.
- Provide a high degree of control and take input on how the map should look.
- Be able to iteratively make changes to a map based on the user's direction.
- Be highly expandable and changeable to accommodate changes in the game that uses its maps.
- Be relatively quick, and scale well with larger maps.

- Be able to generate a map of given dimensions all at once, though chunk-based generation should not require too many changes to the code.

Creating a single generator that can satisfy all these requirements at the same time is unrealistic. This is why we decided that instead of creating a single generator, we would create a framework that can be used to put together the ideal generator for the situation. This allows the user to tailor the generator to their specific priorities and needs.

Project motivation

The goal of this project is to create a useful tool for procedurally generating terrain, whether it be directly integrated with a game or used as a separate tool for map creation. It should be useful for our employer in the development of their game, allowing them to save time and resources when making maps, as well as facilitating more experimentation during development, thus allowing them to more rapidly add new features. It should also be useful for other developers who want a map generation tool, whether they are unfamiliar with map generation and want a quick implementation or want to highly customize it themselves.

This is not a problem exclusive to Progress Interactive, as map design is a slow and expensive process across the game industry. Procedural generation can massively aid in the map design process, and if implemented carefully, can even replace it. For smaller game studios, procedural generation may be the only way to create a massive world with a high level of detail. Our program could be a tool for many developers to fully realize their games.

This project is especially interesting for us due to our interest in Unity3D and graphics programming in general. Procedural generation is also an interesting topic that we have seen in many forms of media, but never made ourselves.

Procedural generation components and theory

Procedural content generation

Procedural content generation refers to using algorithmic means to create game content procedurally [1]. This typically means that content can be generated on the fly using a small number of parameters or seeds, which can be randomized. The advantages of this include reduced costs, as content need not be manually created every time, and reduced memory consumption, as complicated assets can be represented with little data until it is needed. In this paper, we focus mainly on the former. This is because one of the main goals of our employer was to reduce development costs for his game. The program also needed a pre-defined map size before generating, meaning it does not need to infinitely generate content during runtime, and can instead save the generated map to a file to be loaded later.

In this paper, we will refer to this simply as procedural generation, or procedural terrain generation. We are using procedural generation specifically to create terrain maps to be used in games, including the real-time strategy game of our employer.

Ways to represent terrain

A heightmap is a 2D grid of altitude data. Each point in the grid represents a 2D coordinate for the terrain, and the value represents the altitude this point should have. This is a simple and efficient method, as each tile only needs a single float value to represent the altitude at minimum. Downsides include a lack of multiple levels of detail, and no support for complex terrain like overhangs, caves, or floating islands, as each tile can only have one altitude value [2].

Layered heightmaps are another option. They can be used to represent multiple layers of materials, like rock, sediment, or water. Each layer is essentially a separate heightmap for a separate material, and these are stacked on top of each other in a certain order. This is useful for simulating erosion or digging [3].

A voxel grid is a 3D matrix, where each coordinate either has air or a type of ground. This allows for more complex shapes, such as caves or floating islands. It is however computationally expensive, and is especially taxing for the memory of a game, both when storing the data to file and loading it at runtime [2].

A vector displacement field can be a compromise between heightmap and voxel grid, as it allows for overhangs and caves without needing a 3D grid. It uses a 2D grid of altitude values like a heightmap, which can be thought of as offsets in the Y direction, but each point also has two offsets in the X and Z direction. This means the point in the terrain mesh that is gets used by the terrain mesh can be tilted slightly to the side, allowing for overhangs and even caves. This however does not allow for caves with multiple entrance or floating islands, as topologically any vector displacement field is the same as a plane [4].

We chose to use a heightmap to model our terrain. The map data would consist of a 2D grid, with each tile having an altitude value, as well as other data like biome type or resources present. The reasoning for this was based on the type of game the employer was developing. A real-time strategy game typically has a top-down perspective, with little need for more complex geometry like caves or floating islands. This format is also simple yet flexible, as users can add many different kinds of values to the tiles to represent more abstract attributes. A user could also add multiple altitude values and use them like a layered heightmap. Using a 3D grid would however require more significant changes to the code.

Infinite procedural generation

Procedural generation is often used to create infinite worlds. As the player moves around, the world only generates within their line of sight, and unloads as they move away. This requires consistency, as when the player returns to a location they previously explored, it should be identical to the first time they saw it. To achieve this, the terrain has to generate in a deterministic way, that is unaffected by the terrain's surroundings. Minecraft is an example of a game that uses this method [5]. Each "chunk" of the world can be generated without needing to know what the surrounding chunks look like, and yet the chunks blend together seamlessly. This is because the factors that are used to build the terrain are all part of the same smooth noise functions. Two nearby coordinates in the noise function will be close to each other, without needing to know what each other's values are.

We experimented with this concept early on, using multiple Perlin maps to generate the factors that would determine a tile's biome. A tile's biome could be found with only the coordinates and the seed. However, as we eventually decided maps would have a pre-determined size, this requirement was dropped, though the Perlin function still has this functionality. Our program could be modified to allow for infinite world generation, by using a grid to represent a chunk instead of the entire map.

Cellular automata

Cellular automata are models that consist of a grid of cells that each contain one or more value. The cellular automata changes its state in discrete time steps, where each cell's next state is determined only by nearby cells, or its "neighbors" [6]. We can use cellular automata rules when spreading biomes. Each tile checks its surroundings, and based on the nearby biomes, its own biome may be overwritten. With more complex rules, we can simulate biomes spreading at different rates and competing with each other as they overlap. Cellular automata are also a useful tool for simulating dispersion. If for example we have a temperature map with an unnatural distribution, cellular automata rules can spread the temperature around in a more smooth pattern, similar to how temperature disperses in reality.

Lazy flood fill

Lazy flood fill is an algorithm that affects tiles in a grid in a random "blob" shape, spreading from a given position [7]. It could for instance be used to set all tiles in the blob to a specific biome. The algorithm begins with a queue that only contains the starting tile. While the queue is not empty, a tile is popped from the queue and the change is applied to it, in this case setting its biome. At this point there is a chance that all of the tile's neighbors are added to the queue. This chance gets smaller every time a tile is visited. The rate at which this chance gets smaller is determined by the decay factor, with less decay resulting in larger blobs.

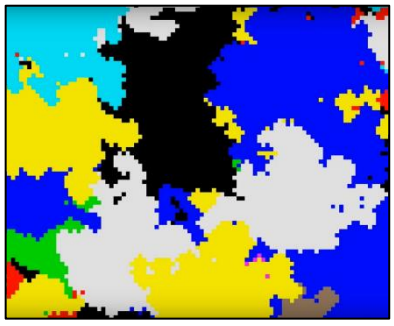


Figure 1: Image of blobs produced with lazy flood fill. Source: https://www.youtube.com/watch?v=YS0MTrjxGbM

Perlin

Perlin noise is a noise algorithm created by Ken Perlin [8]. It creates a noise value for any given coordinate. It does this by considering all coordinates whose x and y values are integers, also called lattice points, surrounding the given coordinate. Each lattice point has a pseudo-random vector associated with it. For each surrounding lattice point, the dot product of the pseudo-random vector and the distance vector to the given coordinate is found. The resulting vectors are interpolated to find the final noise value.

This is only one iteration of the Perlin noise function. To get more interesting terrain with varying levels of detail, we can use multiple iterations, called octaves, with varying parameters. Each octave has its own "offset". This is essentially just a displacement along the noise map, to ensure no octaves have any overlapping patterns. In addition, "lacunarity" and "persistence" values can be defined to determine how much the "frequency" and "amplitude" changes every octave respectively. Greater lacunarity means the frequency changes more rapidly between octaves. Greater frequency means the Perlin noise

map will be more “zoomed in” or have a higher “roughness”. Therefore, high lacunarity means subsequent octaves will have a greater level of detail. Lower persistence means the amplitude will decay more quickly between octaves. Higher amplitude means the noise value is weighed more heavily when the values of all the octaves are added together. Therefore, lower persistence means subsequent octaves will be less impactful to the overall noise value.

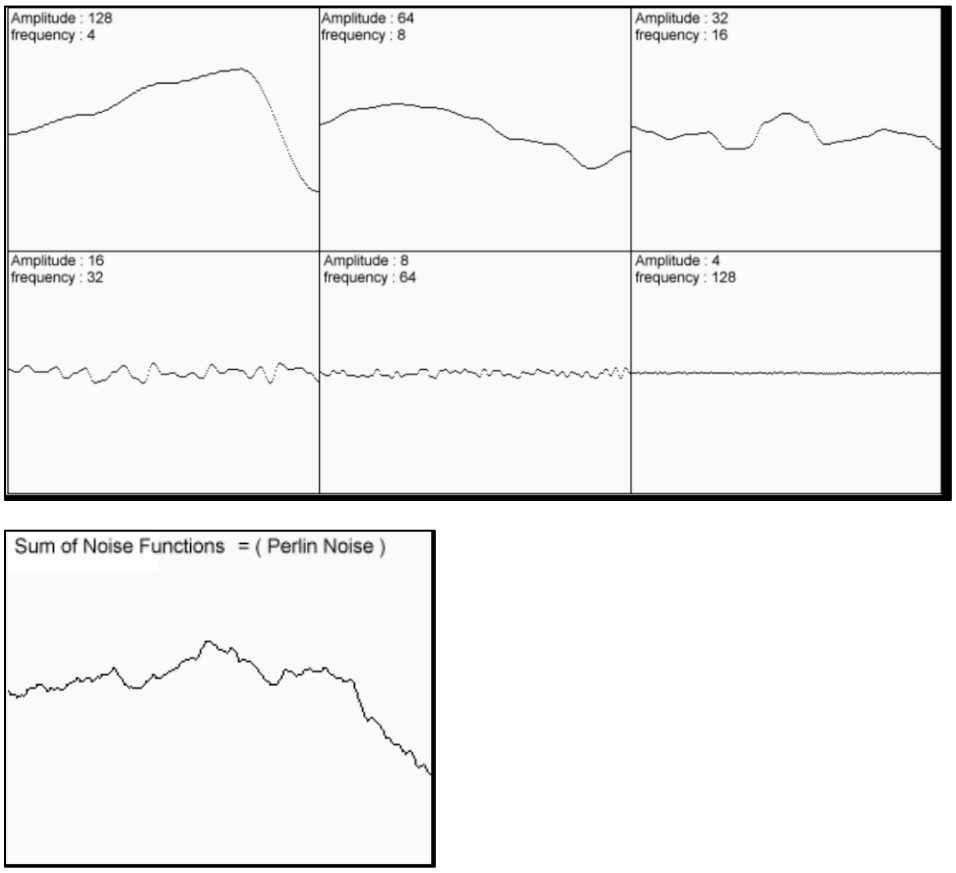


Figure 2: Octaves with various amplitudes and frequencies and the sum of the noise functions. Source: https://adrianb.io/2014/08/09/perlinnoise.html

Diamond Square

The Diamond Square algorithm is a noise algorithm which can be used to generate heightmaps [9]. Unlike Perlin noise, the entire map must be generated ahead of time, and with a specified size. The generated heightmap must be a square of side length (2^n)+1. It can use a base heightmap with a lower resolution as a sort of “seed” to determine the general shape of the more detailed heightmap. Alternatively, it can simply randomly pick the values of the 4 corners as a starting point to get a random shape.

The process begins with an outline of values, whether that be the 4 corners or a more complicated pattern. The values that have not been set are filled in with a series of alternating diamond steps and square steps. The square step goes through each square in the grid, takes the average of the 4 tiles that form the square and sets the tile in the center of the square to that average, plus a random value. The

diamond step similarly goas through each diamond in the grid, and sets the midpoint of the diamond to the average of the 4 corner tiles plus a random value. The range of the random values typically get smaller for each iteration of the steps.

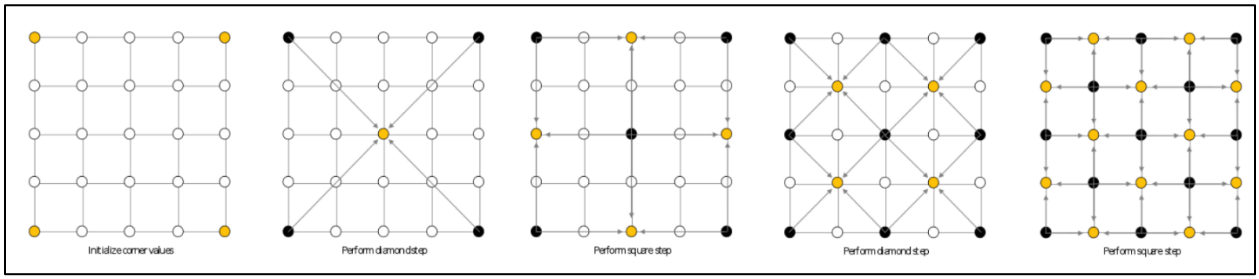


Figure 3: A diagram showing the diamond and square steps. Source: https://en.wikipedia.org/wiki/Diamond-square_algorithm

Technologies

C#

C# is an object oriented programming language developed by Microsoft in 2000. It was originally almost identical to Java, but has over the years developed in a different direction. While C# was originally closed source and heavily Windows dependent, it has since become open source and started to rely less on the Windows OS [10].

Unity

Unity is a game development engine for 2D and 3D games, but can also be used in other fields like simulation. Unity uses C# for scripting. Unity sells licenses depending on the organization using it, but a free version can be used for testing and small developers with revenue under $100k [11].

Binary file

Due to the way binary data is organized, you often have to use more space than you need. For example, if you need to represent an on/off switch, you only need one bit, but you still have to reserve a whole byte. While this often is not an issue, in some cases the space used becomes important.

One way to reduce wasted space is to reserve one byte and store one Boolean value on each bit, that way you can cut 8 bytes down to 1. The downside is that to read or write to each value, you have to do several steps, costing more performance. This is a major cost, but when storing or transporting data it can often be worth it.

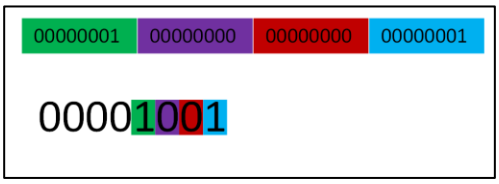


Figure 4: 4 different Boolean values in separate bytes and how they'd look stored in one byte together.

One example of this is Discord's API for bots. When sending out permissions for a role or user, they pack all the different permissions into one big number [12].

JSON

JSON stands for JavaScript Object Notation and is a data modelling language that focuses on portability. JSON is structured identical to how JavaScript structures its data [13].

The data is organized in objects with key-value pairs. Each key in an object is unique, and the values can also be objects or arrays, allowing for nested data.

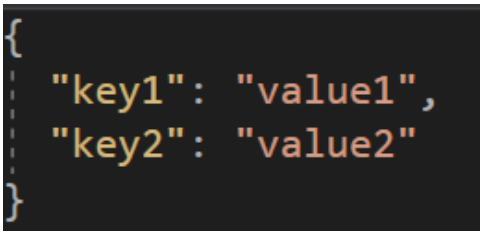


Figure 5: A JSON object with 2 properties.

An object is represented as a pair of curly brackets {}, with properties inside the pair. Properties are represented as a name (key) in quotes followed by a colon, then a value and a comma if there are any properties after it.

For example **"name": "Bob"** is a property with name as the key and Bob as the value.

The value can be a variety of different types:

- A number.
- A Boolean. (true/false)
- A string. (Written between quotes "")
- An object. (Written as a pair of curly brackets {})
- An array. (Written as a pair of square brackets [])
- Null. Representing the absence of any value.

Figure 6: A JSON object with a JSON object in its second property.

JSON supports a variety of common data types, and with objects it can match almost any class's structure providing portability between different languages, while also providing a high degree of readability. This makes it a popular format for storing and transferring data, especially when the recipient's programming language is unknown.

3 Requirement specification

Requirements

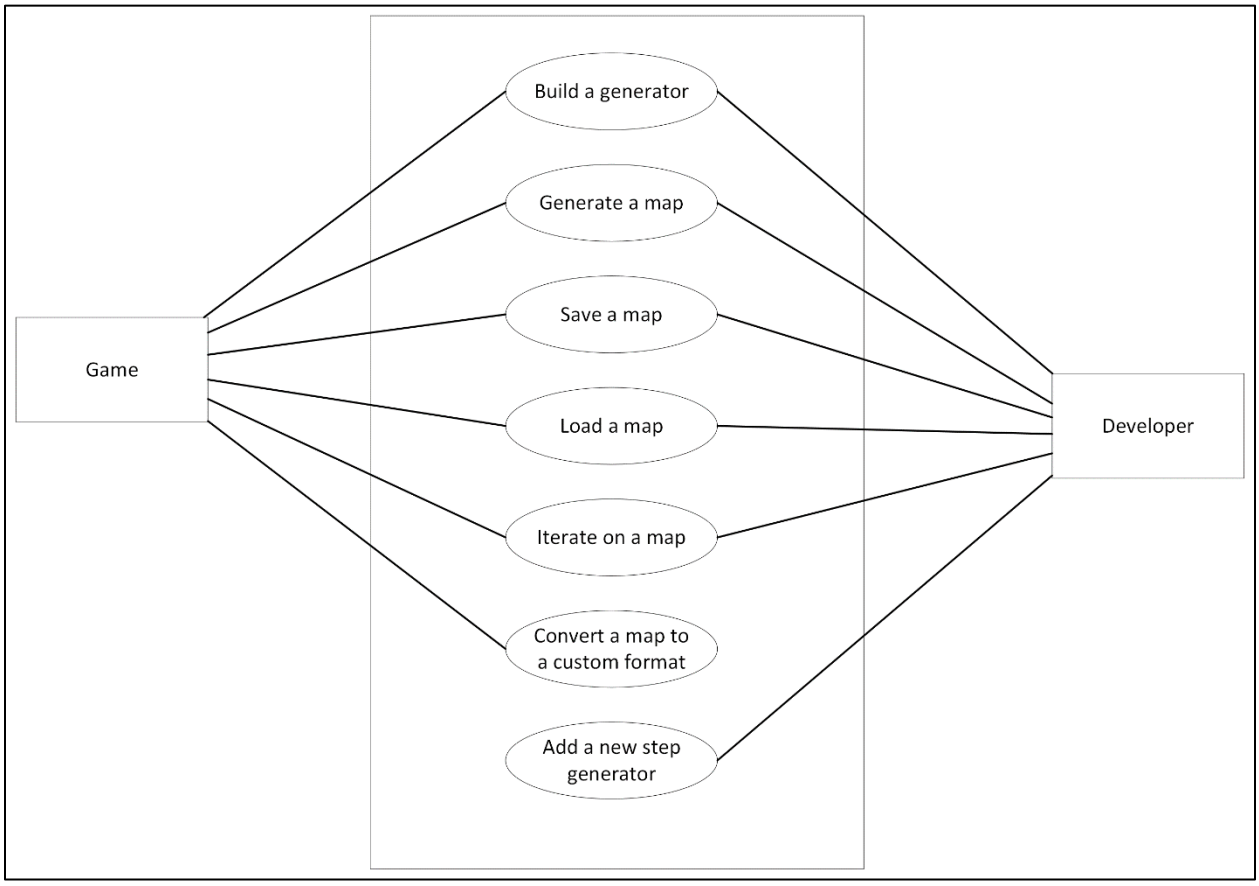
Functional Requirements

- The program must be able to generate maps consisting of rectangular tiles.
- The program must be able to generate maps of different sizes.
- The tiles must be able to contain altitudes, biomes and resources.
- The program must be able to convert these maps into files that it can convert back into maps.
    - Originally, this was to be stored in an RGBA image, but due to limitations with the format, it was agreed to change this.

Non-functional requirements

- The program should be able to generate a map of reasonable size in a reasonable amount of time. For example, a map of size 500x500 should take less than a minute to generate.
- The same generator, with the same configurations and seed, should yield the exact same map each time. (All randomization should be based on the seed, never use system time or global randomizers.)
- Must be able to take input from the user to alter its behavior.
- Must be able to improve an existing map and allow the user to add new steps along the way.

Use-cases


Build a generator
Generate a map
Save a map
Load a map
Iterate on a map
Convert a map to a custom format
Add a new step generator
Game
Developer

4 Design

Overall design and rationale

With our design, our biggest priority was adaptability. With something as complex as map generation, there is no one-size-fits-all solution. Different specific cases need different solutions, but since our employer did not have a detailed specific case, we needed a flexible solution. We focused on making a highly customizable tool that the user could tailor to their needs. Additionally, with our limited experience with procedural generation, we would need to experiment a lot, so a flexible design would be key to development.

The map generation is flexible in the following ways:

- Map generation can be divided into steps, which can each be turned on or off
- Each step has a number of generators the user can choose from
- Each generator has a number of parameters the user can change
- The user can also use existing maps as a basis for generation, even if it was not made with this program, provided it is the right format
- The user can easily code new generators for a given step
- The user can easily code new steps and make generators for them
- The user can easily tweak our existing code to remove restrictions

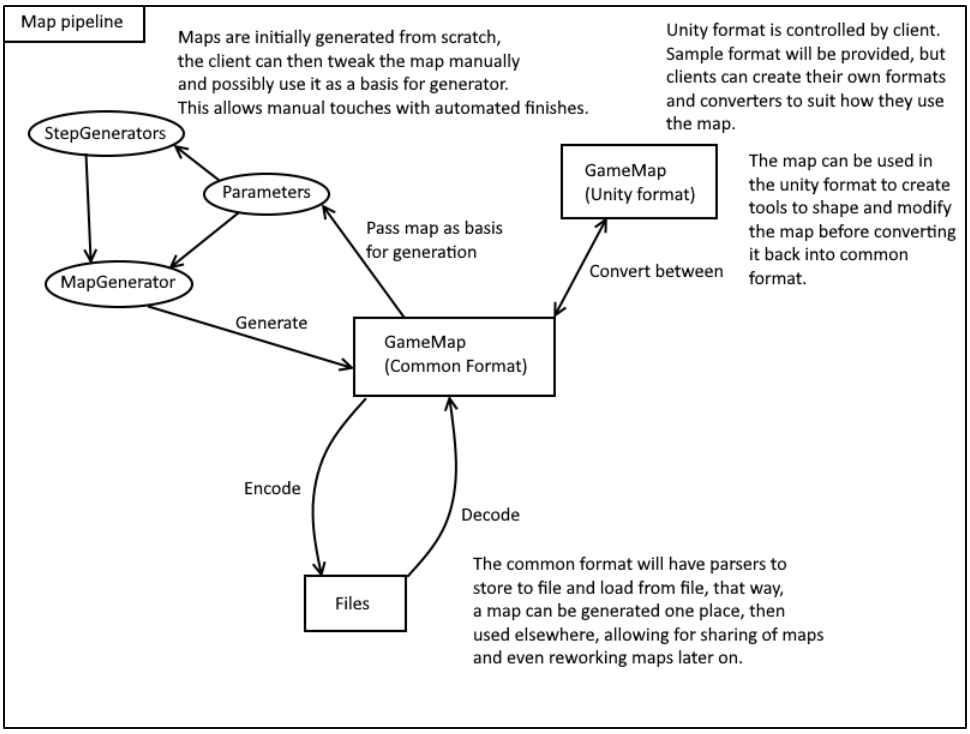
For the program to work, some restrictions must be made, though we put a lot of focus into making as few restrictions as possible and making them as easy to change as possible. These restrictions include things like there only being one biome in each tile, or the map having to be a rectangle. Overall, our design provides a lot of flexibility to fit the user's needs, while also providing an easy-to-follow structure.

While we considered performance and scalability in our design, it was not a major focus. If performance becomes a high priority for the user, they can easily create more optimized versions.

It is up to the user how they want to interpret and use the generated map data. For instance, the altitudes of each tile could be used to create a 3D model capable of physics simulations, or it could be rendered entirely in 2D, with the altitude only being an abstract attribute of each tile that affects the game mechanics.

Overview of the map pipeline

As was outlined earlier in the report, creating a generator that adequately supports every need and every case was unrealistic, especially since the needs change rapidly. That is why we instead focused on creating a tool that was flexible and adaptable enough to keep up with the changing needs.



The diagram above depicts an early design of how the program works. It describes 3 general features that all center on the game map.

1) A map generator is put together and given parameters to describe the kind of map the user wants. Then it is used to create a map. The user can also send in a map to use as the basis, allowing them to improve an existing map.
2) The map can be encoded into a file format, and the files can be decoded into maps, allowing portability.
3) The map can be converted into a custom format the developer creates, allowing them to use it in their game or in other tools that let them represent and manually alter the map. They can convert it back to then run it through more generators or save it.

The goal of this design is to provide a flexible standard for our generators, while also allowing developers to integrate it with their game and the specific needs it might have.

Core data structure

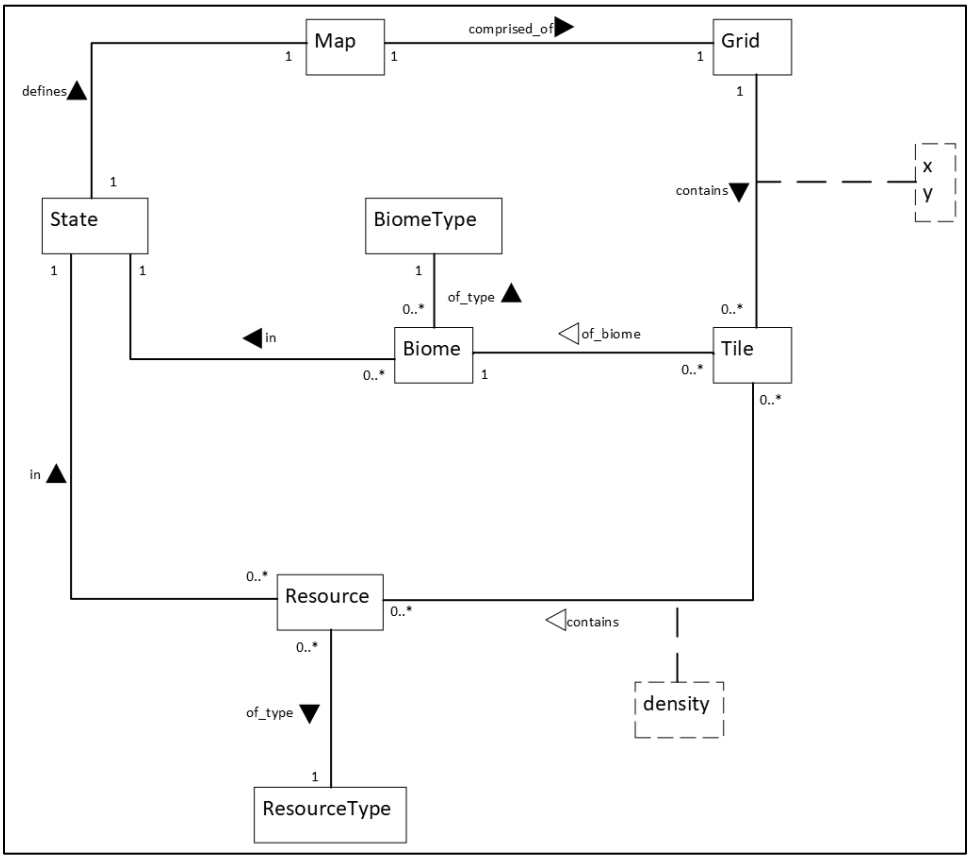


Figure 7: Shows the way different classes are combined to form a map.

Resources

Resources describe entities on a tile that players could take advantage of. Each tile can have multiple resources and they can have different densities on different tiles. The resource types of this project are limited, but users are encouraged to add their own to fit their needs. At this point, the resource types are gold, iron, copper, berries, cotton, and spice.

Biomes

A biome is a general category for an ecosystem. Biomes are used for many aspects of the generation process, like smoothening terrain or picking what kind of resources can be placed where. Each tile can only have one biome.

Each biome has a biome type, informing the program what kinds of rules apply to this biome, but biomes of the same type can differ in other aspects. The code has sub-classes for biomes of different types, allowing users to assign different attributes to biomes of different types. So far, this has only been used for lakes, which need their surface level stored.

The biome types included in the project are:

- Ocean, Mountain, Sand, Stone
- Soil
- Desert, Savanna
- Grasslands, Forest
- Jungle, Swamp
- Tundra, Boreal, Arctic
- River, Lake, Glacier

Tiles

A tile is a square area on the map. Tiles only store information about the tile itself, like altitude, temperature, humidity, wind, resources and biome. Tiles do not store information about where within the tile things are placed.

It is intended that when using these maps in games, the game should have its own map structure that our structure converts into. That way, our generic map can generate the overall design of the world, and the game then translates that into the format they need. For example, they could convert a density of trees into actual trees on the tile, smoothen the borders between tiles to make it less blocky, or store whatever other information they see as relevant.

The values of a tile are also ambiguous and up to the user to interpret. Things like densities, temperatures, and altitudes are given in values between 0 and 1. That way, the code works regardless of what the ranges mean. For example, an arctic map could have temperatures of -50°C to 5°C, while the temperatures on a generic map could mean -10°C to 40°C. The exact size of a tile is up to the user as well. It could represent anything from 1x1m to 1x1km depending on what the user needs.

Grid

A grid is a 2D matrix of tiles. It contains every tile in the map. The dimensions of the grid can be any width and height, though valid ranges can be defined by parameters.

GameMap

The GameMap is composed of a Grid and a MapState. It contains all the data of a map.

MapState

The MapState describes overall information about the map, such as the size, what altitudes oceans and mountains are, and what biomes and resources the map has.

The step generator model

Instead of developing complete generators that will handle the entire process of generating a map, we decided to break this process into steps and develop generators for each of these steps, referred to in this report as step generators.

These steps are a way to organize and manage step generators, but they are not mandatory. For instance, if the user wants to have several mutually exclusive generators that create altitude, they could be organized into an altitude generation step. However, if they want something more complicated, like a step generator that simulates weather patterns and affect wind, temperature, and humidity, they might want to make it without any specific step generator type.

Each step generator should specify which map aspects they require, and which they add. A map aspect represents a part of the map data, such as altitude or biome. For instance, a step generator that sets the biome of each tile based on its altitude requires the altitude map aspect, and adds the biome map aspect.

A major benefit of this design is that it is easy to expand. A new step generator can be added without rewriting others, and they can be picked and mixed however the user wants. It also allowed us to develop generators much faster, try them out in different ways, and create more readable code. If we developed complete generators that make the entire map at once instead, then adding new generators would require remaking every step of the process, and swapping one step out would also require an entirely new generator.

The biggest downside to this design is that each step generator, being isolated, cannot make many big assumptions about the current state of the map. Which limits what they can do to some degree, and sometimes requires doing the same thing in multiple places. However, we find these downsides to be well worth the benefits.

Additionally, the user can also send a map in as basis for the generator instead of creating it from scratch. This way, the user can take a map, make changes to it, then send it back in to make whatever improvements they want. This makes it possible to improve a map step by step until the user is satisfied, instead of generating a map and then manually fixing everything from then on or starting over again.

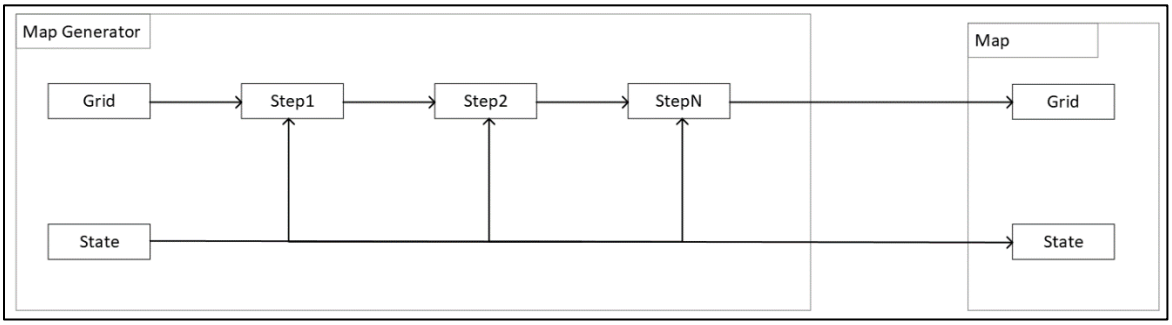


Figure 8: The way map generators pass the map from one step to another.

MapGenerator

The MapGenerators are used to put all the pieces together and generate a map.

All MapGenerators are made up of 3 parts:

1) The GeneratorState.
   This holds information about the state of the generation process as well as parameters that define the map.
2) The StepGenerators.
   These handle different steps in the map generation process.
3) A basis map.
   The basis map is optional, if no map is provided, the generator will generate a new map.
   If a map is provided, it will use it as a starting point and modify it, this is useful if you want to smoothen out an existing map.

There are 2 types of map generators included in the program:

1) MapGenerator
   This model takes in one StepGenerator for each step and runs them in order.
2) ChainMapGenerator
   This is the newer, more flexible model. Instead of following a strict structure for what steps should be done, it takes in all the steps the user wishes to do in the order they want it done, then runs them in that order.
   This model grants the user more control as they can change the order of steps, repeat steps, or even create steps that our program has not considered.

StepGenerator

A StepGenerator is a generator that handles a step of the map generation process, such as river generation or altitude smoothening. The purpose of this is to split the generation process into steps and make multiple generators for each step, allowing users to pick and mix these to build their own MapGenerator. Each StepGenerator has its own configurations, these can be used to control how each specific StepGenerator operates. These configurations can be used for things like controlling how far to search/spread, how many iterations to run, or turning features on or off.

We have implemented the following types of StepGenerators, listed in the default generation order:

- IAltitudeGenerator — set altitudes
- IAltitudeSmoothenerGenerator — smoothen altitudes
- IOceanMountianGenerator — set ocean and mountain level
- IRiverLakeGenerator — make rivers and lakes
- IWindGenerator — set wind
- ITemperatureHumidityGenerator — set temperature and humidity
- IBiomeGenerator — set biomes
- IMapSmoothenerGenerator — smoothen the map
- IResourceGenerator — set resources
- INatureGenerator — make nature (no generators)
- IArtifactGenerator — make artifacts (no generators)

A StepGenerator does not need to be of a specific type. In this case, it can only be used by ChainMapGenerator. MapGenerator requires each step to have a designated type, and can only have one generator of each type.

GeneratorState

This object maintains information about the state of the generation process as well as parameters that define the map. Examples of state information are what biomes and resources the map contains and the altitude of the ocean and mountains. Examples of parameters are size, name and how to store the map.

Map saving/loading system

The maps can be saved to files seamlessly, which can then be loaded back into the program later.

Saved files

When saving a map, a folder with the map's name is created in the desired folder. Within this folder, 3 files are created:

1) State.json
   This file contains information that applies to the entire map, such as size, biomes and resources on the map, and how the tiles are encoded.
2) Grid.map
   This file contains the information about each tile in the map, such as altitude and biome.
3) Thumb.png
   This is a thumbnail that shows users how the map looks.

How tiles are encoded

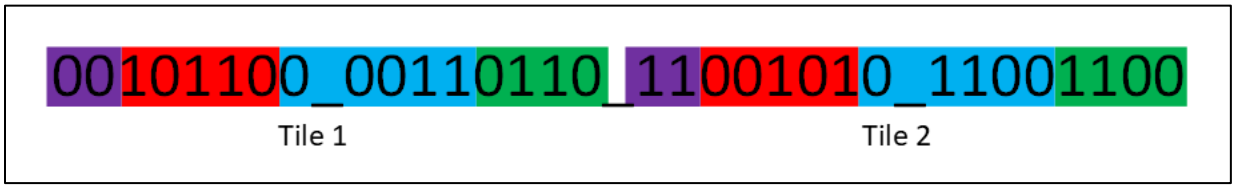


Figure 9: Different tile properties spread around bytes. Each color represents a tile property.

When encoding the map, each tile is given a specific amount of space each where they store their information. They are then written in order into the file. Each property of the tile is given a specific number of bits to use, and the property is compressed to fit that space. The more bits a property is given, the more precise it is saved. This way, the file is as small as possible.

Configurations of tile encoding

Each map has configurations to control what properties of a tile to store and how many bits to use on each. This allows the user to control what they want to use space on and how big the file gets. If precise altitude is important, the user can increase the number of bits to use for altitude. If the map does not use wind, then wind can be omitted, reducing the file size. This gives flexibility in how precise the values are stored, and allows the user to minimize the file size where that is important.

The simple testing GUI

In our program, we also include a GUI that we used for testing in the Unity editor. This GUI is not advanced, and it only provides basic features that can be used to play with the generators and see what it makes. It is composed of 2 scenes.

The first scene runs scripted tests when started up. This is useful for more complex tests that do not require user input, such as benchmarking or testing the encoding/decoding.

The second scene can visualize a map in 3D, allowing for more detailed inspection of how the map looks than the 2D images our code can generate. Additionally, it allows users to customize their own generator and run it, displaying the results in 3D when done. This is more practical for trying out different generators and configurations and seeing what gives a more desirable result.

Figure 10: Screenshot of the testing GUI.

5 Development process

Work process

We organized our work process into one-week sprints from Monday to Friday. This allowed us to quickly adapt to changes. The work hours were agreed upon at the start of each sprint. This gave us a consistent schedule to organize our work around, while also allowing us to change it to adapt to the situation. Generally, our work hours were from 09:00 to 16:00, but some weeks we increased or decreased it. We decided the two of us would work together in a physical location. This would help put our mind into work mode and would facilitate rapid communication as things happened. All our work hours are documented in the time log.

Changes in specifications

Before writing any code, we focused on defining and breaking down the specifications of the project. The employer was open to keeping the project flexible, and we had several discussions and brainstorming sessions about what we could integrate into their game. We decided the first thing we would develop was procedural map generation. We initially expected this to be only part of the project, with other ideas including placing buildings and a unit and resource management system. This would require working closely with the employer, and using and expanding on their existing code. However, shortly after our project started, the employer started a new full-time job, resulting in significant scheduling conflicts. Although we were available at practically any time of week, the employer rarely had time for meetings, and we were not able to work on the company's code at all.

The lack of meetings, communication, and feedback meant the specifications of the project had to change significantly throughout the development process. We eventually decided, along with the employer, to dedicate the entire bachelor to procedural terrain generation, as this could be done independently of the game itself. We would still tailor the project to the employer's needs, but the only connection to their code would be a file representing the terrain that could be read by the game, as well as any other game.

The specifications we did have from the start for the employer's game were regarding the map dimensions, altitudes, and biomes. The tool would have to generate a map with given dimensions all at once, as opposed to generating infinitely or loading only certain parts of the map at a time. The map would need to be a 2D grid, where each coordinate, or tile, would have a value representing its altitude and a biome. The altitudes would need to look smooth and natural, and the biomes would have to look like blobs of adjacent tiles. These specifications were kept throughout the entire project.

Methodology

We originally wanted to follow a more rigid methodology. The plan was to have weekly sprint meetings where we would decide what issues we wanted to complete that sprint. These issues would be selected from a backlog of features we would make with the employer. This backlog would contain more features

than we were likely to implement in the given time, sorted by priority. We wanted to spend the first few weeks planning and discussing with the employer while making this detailed backlog.

When we realized long term specifications would not be possible, we instead adopted a more flexible methodology. We would discuss what features were most important to implement first between the two of us, and make short-term plans for the rest of the week. Since we were only two members, this and communication during our work hours was sufficient. Despite this flexible structure, we had rigid working hours, and we were able to consistently implement features. We initially focused on making a functional MVP as fast as possible, then improved the parts that seemed the most lacking from there. We always maintained a long term perspective, and focused on making the code modular and expandable. This made it easy to add more features or change existing ones with minimal extra work, should the specifications change. There was also a strong focus on research, as we experimented with different methods, such as when generating altitudes, or simulating wind. Sometimes we would find a different generation method than one we had already implemented but kept the old method in the form of another step generator, as both could be useful to the end user.

Tools and platforms

- BitBucket – Repository.
- Visual Studios – Programming.
- Unity – Development and testing.
- Sharepoint – Hosting documents.
- Microsoft word – Documentation.
- Microsoft excel – Timesheets.
- Discord – Communication and digital meetings.

Repository

We had all our code stored on one repository, owned by the employer. We generally worked on separate branches when implementing new features, making sure not to both of us commit to the same branch. We would merge these branches with dev when we were finished with significant features, or when one of us needed the changes the other had made. We made sure we were confident that the code was stable before merging, and always communicated with each other before doing so. We often used peer programming methodology at this stage to make sure the merge went cleanly and to fix any merge issues. We only pushed changes directly to dev when fixing issues after a merge, when adding comments, or when making quick changes we both agreed were necessary. In this final case, we always used peer programming.

Commits were of a medium size. Sometimes a quick fix or simple new addition to the code would take a few minutes, and other times large features would take several hours. This was because we did not want to commit code that could not be compiled, and some large features or overhauls required a lot of time to get working. In these cases, we always underwent rigorous testing and used peer programming. Our commit messages were always short and informative, though the branch names were sometimes vague or became outdated.

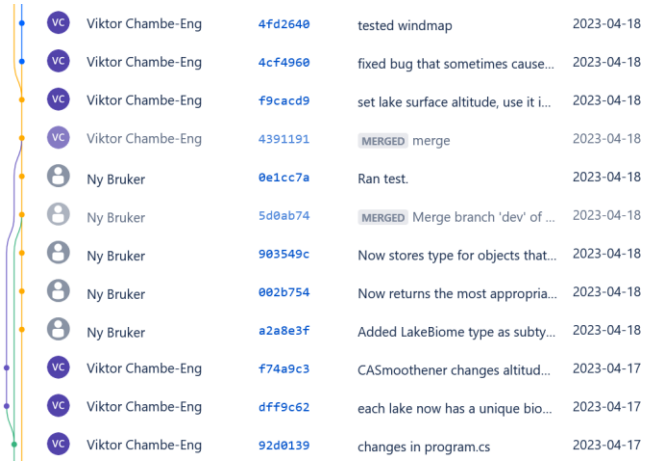


Figure 11: An example of our commit history.

6 Implementation

Core

The core code is the code that defines the fundamental structures of our program. This code defines how our map data is structured.

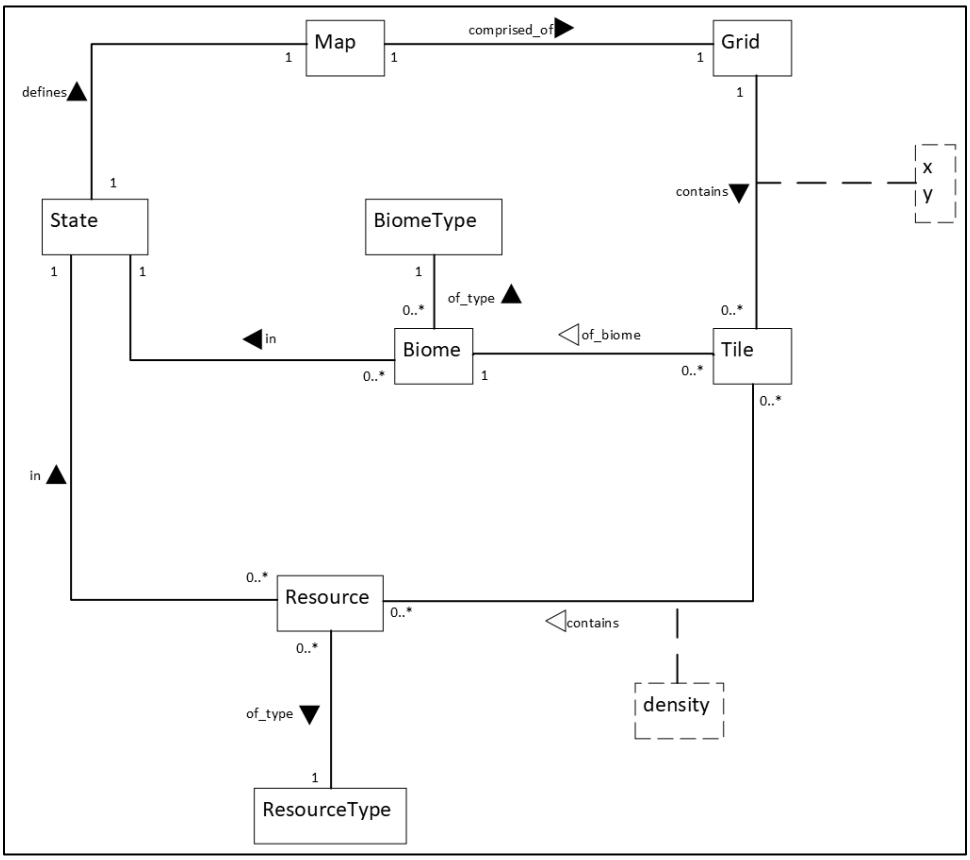


Figure 12: The class diagram of the classes that make up the main data.

Resource

Each map can contain resources. These could for example be used by players to perform certain actions or make things. Our program makes as few assumptions about these resources and their use as we can.

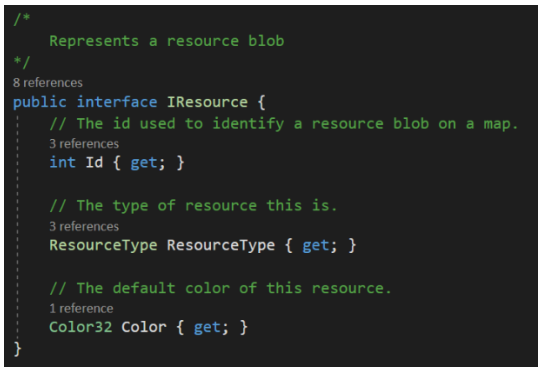


Figure 13: The IResource interface.

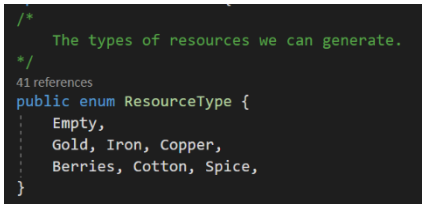


Figure 14: The ResourceType enum.

Each resource is its own instance, and additional properties about these can be added with few changes. So far, we only store id and the type of resource it is. Further, each resource can be present on multiple tiles, and has a density to represent how much of it is on the tile. We have only included a simple set of resource types, but these can easily be expanded to fit whatever needs the game has.

Biome

A biome represents the ecosystem present on a tile.

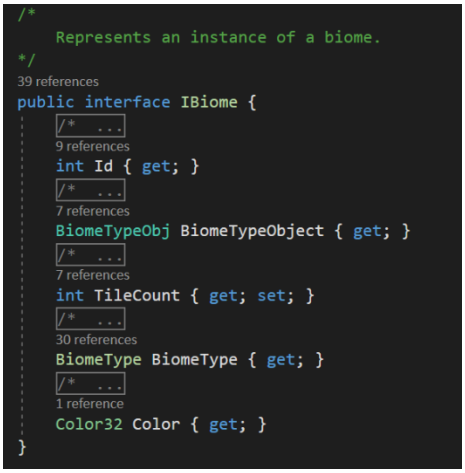


Figure 15: The interface for biomes.

Each biome is its own instance and can contain information about that specific biome. Each biome is of a biome type, but biomes of the same type are still separate, allowing us to for example make different forests work differently.

```
/*
    The type of biomes.
*/
99+ references
public enum BiomeType {
    Empty,                          // Used for tiles with no specific biome defined.
    Ocean, Mountain,                // Used for altitude extremes.
    Sand, Stone,                    // Sand is a buffer for ocean, stone is a buffer for mountain.
    Desert, Savanna, Jungle,        // Hot biomes.
    Grasslands, Forest, Swamp,      // Mid biomes.
    Tundra, Boreal, Arctic,         // Cold biomes.
    River, Lake,                    // Water source.
    Soil, Glacier,                  // Depricated.
}
```

Figure 16: The types of biomes.

There are different types of biomes.

```
/*
    Represents an instance of a lake biome.
*/
6 references
public interface ILakeBiome : IBiome {
    /*
        The altitude of the water surface of the lake.
    */
    6 references
    float SurfaceAltitude { get; set; }
}
```

Figure 17: The interface for lake biomes. Contains an additional property.

The biomes can be extended with additional properties if more detail is desired, allowing developers to fit the system to their needs. Additionally, there are sub-classes for biomes of different types, allowing developers to add properties to biomes of certain types without having to add them to all types. The only use of this in our code so far is to let lakes keep track of where the surface of their water is.

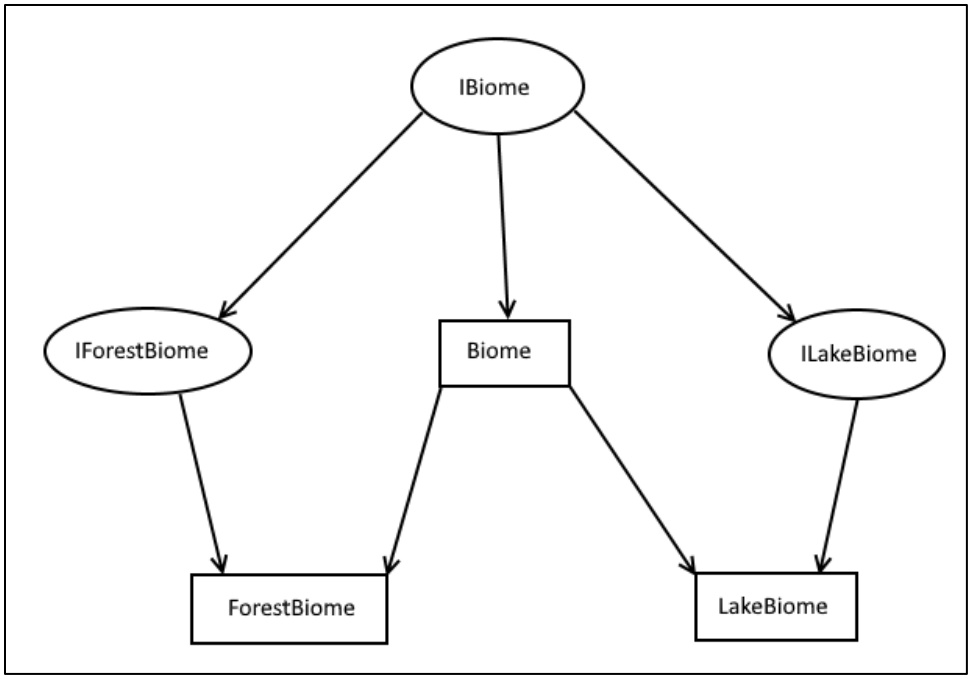


Figure 18: Diagram showing iInheritance relationships for the Biome classes. (Using forest and lake as examples of sub-classes)

Tile

Tiles are the building blocks of a map. Each tile represents an area of the map and contains properties that differ between each tile.

```
public class Tile {
    public const int NUM_ALT_GROUPS = 4;          //
    public const float OCEAN_ALT_CUTOFF  = 0.4f;  //
    public const float MOUNTAIN_ALT_CUTOFF = 0.8f; //

    // The biome of the tile.
    public IBiome Biome;

    // The altitude of the tile.
    public float altitude;

    // The temperature of the tile.
    public float temperature;

    // The humidity of the tile.
    public float humidity;

    // The x component of the wind on the tile.
    public float windXComponent;

    // The y component of the wind on the tile.
    public float windYComponent;

    // The angle of the wind on the tile given in rad
    public float windAngle;

    // The magnitude of the wind on the tile.
    public float windMagnitude;

    // The age of the tile, used for spreading biome.
    public int age;
```

Figure 19: Part of the Tile class.

Factors like altitude, temperature, humidity and wind strength are stored in tiles as decimal numbers between 0 and 1 representing 0% to 100%. This format is flexible and easy to work with, allowing generators to use and change them without having to know what they mean. The exact meaning of these values can be applied later when relevant.

Tiles also keep track of their biome. Each tile belongs to one biome, and this is used in different parts of the program. The tile stores a reference to the exact biome it belongs to, so if any changes happen to it, all relevant tiles will know.

Tiles also track a list of resources they contain. Similar to biomes, the tiles store a reference to the resource, so that they stay updated on changes. Resources do however differ from biomes in that the tile can have multiple resources at once, and that each resource on the tile has a density, allowing us to control how much there is on each tile.

Map

The game map is composed of a grid and a state.

```
/*
    Class for a game map.
*/
28 references
public partial class GameMap : IGameMap {
    /*
        The grid containing all the tiles of this map.
    */
    29 references
    public Grid Grid { get; private set; }

    /*
        The overall state of this map.
    */
    32 references
    public MapState State { get; private set; }
```

Figure 20: The GameMap class.

Grid

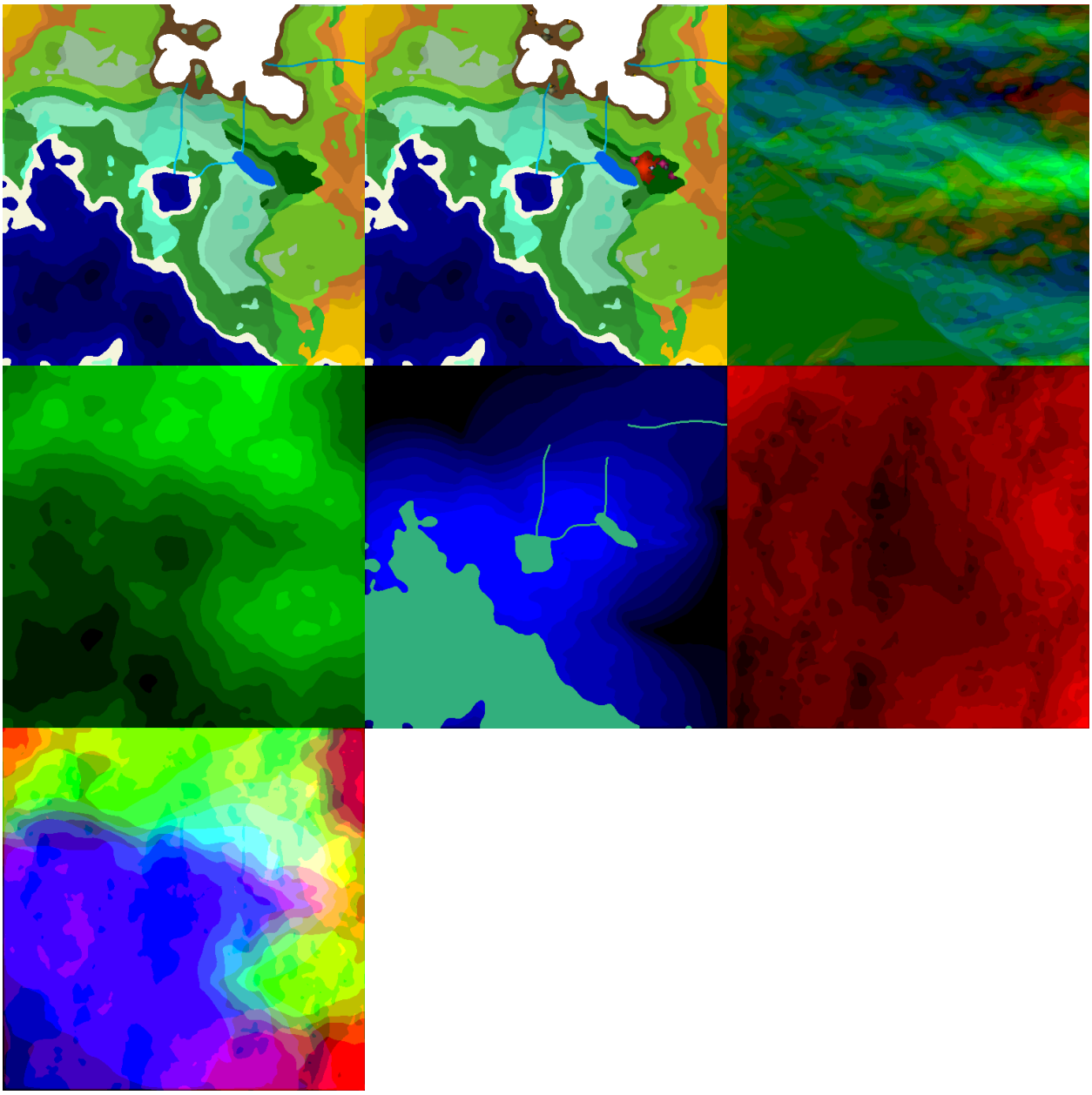
```
/*
    Grid containing all tiles in a map
*/
public class Grid : IGrid {
    Tile[,] grid; //grid of tiles
    int w, h, l;  //width, height, length

    /* ...
    public Grid(int w, int h, IBiome biome = null) ...

    /* ...
    public Tile this[int x, int y] ...
    public Tile this[(int,int) pos] ...


    /* ...
    public Tile this[int i] ...


    /* ...
    public int Width => w;
    /* ...
    public int Height => h;
    /* ...
    public int Length => l;
```

Figure 21: Part of the Grid class.

```
/* ...
public Texture2D ToPrettyImage(float oceonCutOff = Tile.OCEAN_ALT_CUTOFF, float mountianCutOff = Tile.MOUNTAIN_
    => ToImage(BiomeExtension.BiomeColorPallet_Updated, oceonCutOff, mountianCutOff);

/* ...
public Texture2D ToImage(IReadOnlyDictionary<BiomeType, Color32> colorPallet, float oceonCutOff = Tile.OCEAN_AL


/* ...
public Texture2D ToAltitudeMap() ...

/* ...
public Texture2D ToTemperatureMap() ...

/* ...
public Texture2D ToHumidityMap() ...

/*
    Creates a map colored by the 3 main factors.
*/
public Texture2D ToFactorMap(
    float biomeCutoffA = GenerationUtils.BIOME_CUTOFF_A, float biomeCutoffB = GenerationUtils.BIOME_CUTOFF_B,
    float biomeCutoffC = GenerationUtils.BIOME_CUTOFF_C, float biomeCutoffD = GenerationUtils.BIOME_CUTOFF_D,
    float oceonCutOff = Tile.OCEAN_ALT_CUTOFF, float mountianCutOff = Tile.MOUNTAIN_ALT_CUTOFF,
    bool drawExtremeAltitudes = false
) ...

/* ...
public Texture2D ToWindMap() ...
```

Figure 22: Some methods in the Grid class for creating images.

A grid is composed of tiles, each with an x and y position. The grid represents the layout of a map. It also contains several methods to create images representing the map. These methods highlight varying attributes the tiles contain.



Above are 7 different prints of the same map. In the order presented they are:

1) Pretty print: This shows the biomes and adjusts brightness based on altitude.
2) Resource map: Shows the same as pretty print, but with resources drawn on top.
3) Wind map: Shows wind magnitude as green and direction as red for up and blue for down.
4) Altitude map: Shows the altitude as green.
5) Humidity map: Shows humidity as blue. Humidity sources, like oceans, rivers and lakes are shown as cyan.
6) Temperature map: Shows temperature as red.
7) Factor map: Shows the 3 factors: temperature, altitude and humidity as red, green and blue respectively. This one is particularly useful for seeing how the factors affect things like biomes.

MapState

```
/*
    Interface for the state of the map.
*/
28 references
public interface IMapState {
    1 reference
    string Version { get; }
    18 references
    string Name { get; set; }

    15 references
    float OceonLevel { get; set; }
    15 references
    float MountianLevel { get; set; }

    8 references
    int Width { get; } //width of grid
    6 references
    int Height { get; } //height of grid
    2 references
    int Seed { get; } //random seed

    // Controls what tile properties are saved and how many bits they each get.
    5 references
    IReadOnlyDictionary<TileProperty, int> TileStorageConfig { get; set; }

    /*
        Scaling factor for the map.
        NOTE: MapScalingFactor is not magical, each step generator needs to manuall
    */
    2 references
    float MapScalingFactor { get; }
    3 references
    MapAspects MapAspects { get; }

    1 reference
    IReadOnlyDictionary<int, IBiome> Biomes { get; }


    2 references
    IBiome GetBiome(int id);
}
```

Figure 23: The interface for map state.

The map state holds important information about the map that is not tied to any specific tile.

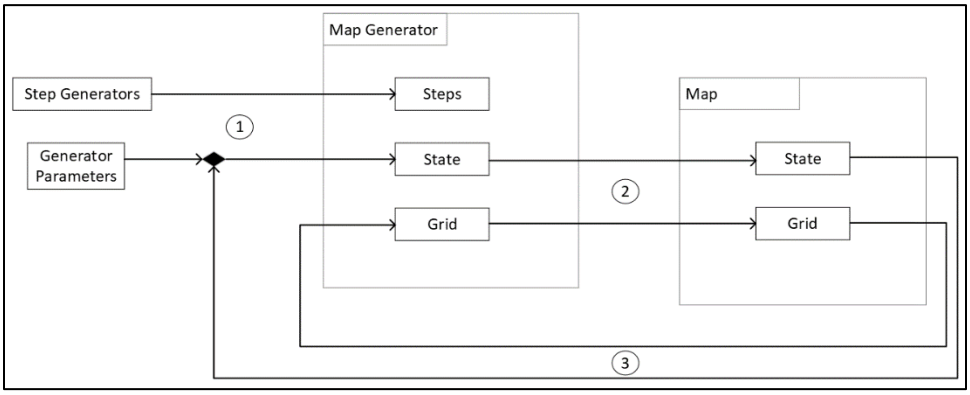
It holds the parameters that define the map, such as name, map size and general rules for the generation process. One such parameter, MapScalingFactor, is used to tell the steps to adjust their configurations based on map size, such as scaling the number of resources. This way, parameters do not always have to be altered when changing the size of the map size. It also contains information related to how to store the map, such as how much space to use for each property of the tiles.

Another important role it plays is keeping track of the overall state of the map.
It keeps track of what altitude the ocean level is and what altitude mountains start at, as well as lists of the different biomes and resources present on the map. This is an important role because the biomes and resources present on the map need to be saved separately form the tiles they are present on. Lastly, it keeps track of what aspects of the map have been generated so far, allowing us to ensure that the map fulfils the requirements for a generator before running it.

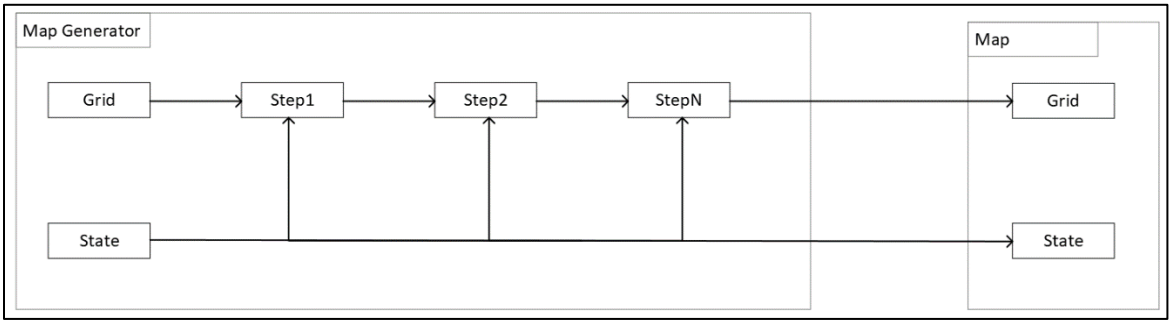
Map Generators

The way maps get generated is through map generators. These generators take in information about the desired map itself as well as what steps the process should go through. If the user wants to iterate on an existing map, they can also send it to the generator.

The pipeline



1. The process starts with the user picking a set of steps to perform and parameters for how the map should look. This is used to create a generator. The parameters are used to create the generator state. A new, empty grid is created.
2. The generator is executed and a map is created from the process.
3. The user can use the map as a basis for a new generator. That way, they can perform new steps as they see fit. When this is done, the state of the map is used together with the generator parameters to create the generator state, and the grid of the map is used instead of a new one being created.



The grid is passed from one step generator to the next one, and the result of the last step generator is the grid of the finished map. The state of the generator is sent to each step generator and is used to create the map state.

```
/*
    The different aspects of a map. (Flags)
    Used for telling what aspects have been added or need to be added.
*/
[Flags]
99+ references
public enum MapAspects {
    None                = 0,
    Altitude            = 1,
    Wind                = 1 << 1,
    TemperatureHumidity = 1 << 2,
    Biomes              = 1 << 3,
    Nature              = 1 << 4,
    Resources           = 1 << 5,
    Artifacts           = 1 << 6,
    OceanMountain       = 1 << 7,
    RiverLake           = 1 << 8,
}
```

Throughout each step, the map generator keeps track of what aspects of the map have been generated and uses this to check if it fulfils the requirements to run the next step.

The implementations

We have two implementations of map generators, one is simply named MapGenerator, the other is named ChainMapGenerator.

MapGenerator

MapGenerator takes one step generator of each type and runs through the different steps. It does not need a step generator of each type, but no more than one of each can be sent.

```
var map = gen.Generate();

var map2 = gen
    .GenerateAltititude()
    .SmoothenAltititude()
    .GenerateOceanMountianGenerator()
    .GenerateRiverLake()
    .GenerateWind()
    .GenerateTemperatureHumidity()
    .GenerateBiomes()
    .GenerateSmootheness()
    .GenerateNature()
    .GenerateResources()
    .GenerateArtifacts()
    .GetMap();
```

```
/*
    Generates the map in the standard pipeline.
*/
7 references
public GameMap Generate() {
    GenerateAltititude();
    SmoothenAltititude();
    GenerateOceanMountianGenerator();
    GenerateRiverLake();
    GenerateWind();
    GenerateTemperatureHumidity();
    GenerateBiomes();
    GenerateSmootheness();
    GenerateNature();
    GenerateResources();
    GenerateArtifacts();



    return GetMap();
}
```

If the Generate method is called, all steps are executed in the default order. However, each step can be manually executed by calling their respective method, allowing the user to run them in a different order or repeat a step multiple times.

```
var gen = new GeneratorBuilder(par) {
    AltitudeGenerator = new PDSAltitudeGenerator(2, 9),
    AltitudeSmoothenerGenerator = new AltitudeAverageSmoothener(3, 3, false),
    RiverLakeGenerator = new RiverLakeGenerator(4, 2),
    OceanMountianGenerator = new BalancedOceanMountianGenerator(0.2f, 0.05f, 50),
    WindGenerator = new FlowingWindGenerator(),
    TemperatureHumidityGenerator = new DiamondSquareTemperatureHumidityGenerator(),
    BiomeGenerator = new FactorBasedBiomeGenerator(),
    MapSmoothenerGenerator = new CellularAutomataSmoothener(2, 2),
    ResourceGenerator = new FloodFillResourceGenerator(),
}.Build();
```

Figure 24: A MapGenerator being created using a GeneratorBuilder.

MapGenerators are created using GeneratorBuilders. These are used to assemble your generator using the builder pattern.

ChainMapGenerator

ChainMapGenerator differs from MapGenerator in a few ways.

```
var gen2 = new ChainMapGenerator(new List<IStepGenerator>{
    new PDSAltitudeGenerator(2, 9),
    new AltitudeAverageSmoothener(3, 3, false),
    new BalancedOceanMountianGenerator(0.2f, 0.05f, 50),
    new RiverLakeGenerator(4, 2),
    new FlowingWindGenerator(),
    new DiamondSquareTemperatureHumidityGenerator(),
    new FactorBasedBiomeGenerator(),
    new CellularAutomataSmoothener(2, 2),
    new FloodFillResourceGenerator(),
}, par);
```

Figure 25: A ChainMapGenerator being created, this one does the same as the MapGenerator above.

Firstly, it takes in a list of step generators that it will execute in the order they were provided. This allows users to provide a completely custom order and use different step generators of the same type. This can be useful to for example smoothen the map in multiple times in different ways.

```
var map = gen.Generate();

while(!gen.Finished) {
    gen.Next();
    // Do stuff between each step here.
}
var map2 = gen.GetMap();
```

Secondly, the steps are executed in the order they were sent in. There is no way to call them out of order. Steps can be executed one at a time using the Next method, or all remaining steps can be executed using the Generate method.

```
/*
    Runs through and ensures that all required map aspects are added before they're used.
*/
1 reference
private static bool _isValidSequence(List<IStepGenerator> steps, MapAspects aspects = MapAspects.None) {
    foreach(var gen in steps) {
        var missing = ~aspects & gen.RequiredMapAspects;
        if(missing != MapAspects.None)
            throw new ArgumentException(nameof(steps), $"Step \"{gen.GetType()}\" is missing the following re
        aspects |= gen.AddedMapAspects;
    }
    return true;
}
```

The reason it is executed in the specific order is that when creating the ChainMapGenerator it checks whether the steps, if done in the provided order, will all have the required map aspects. That way, it can alert the user before running any of the steps if it will not work.

Generator steps

Step generator structure

A step generator is a generator that handles a step in the map generation process. By breaking the process down like this, we make it possible to pick and mix generators for different parts and even change the order, repeat steps, or remove steps. The step generators follow a simple but flexible structure that allow users to create new generators that can make practically any modification to the grid. The step generators receive information from 2 different places.

```
public class DiamondSquareAltitudeGenerator : StepGeneratorBase, IAltitudeGenerator {
    [SerializeField]
    [Range(0f, 2f)]
    [Tooltip("How much you want the tiles to deviate from their average.")]
    public float randomness;
```

Figure 26: DiamondSquareAltitudeGenerator has a field for setting its randomness.

Firstly, they have configurations, which are values that the step generator itself holds, telling it how to behave. This can be things like how many rivers to make, how far humidity spreads or how many times to smoothen the terrain. These configurations are different for each step generator, so if two of the same step generator are used in the map generation, they can be told to operate differently.

```
public override Grid Generate(Grid grid, IGeneratorState state) {

    //generate noise map
    var noise = Utils.GenerateDiamondSquareMap(state.Width, state.Height, randomness, Seed);
    //set noise map values to tiles
    for (var x = 0; x < state.Width; x++) {
        for (var y = 0; y < state.Height; y++) {
            grid[x, y].set(noise[x, y]);
        }
    }

    return grid;
}
```

Figure 27: It uses this randomness along with the size of the map, which it gets from the state object, to generate the altitude.

The second type of information a step generator receives is the generation state. This represents the overall state of the map generation and is shared between all steps. This contains both parameters the user sent to define the map itself, like the size, and values used to keep track of the current state, like the altitudes for oceans and mountains, the biomes present on the map, and the resources present on the map.

```
// Map aspects needed for this step generator to work.
4 references
public override MapAspects RequiredMapAspects => MapAspects.None;

// Map aspects this generator applies.
4 references
public override MapAspects AddedMapAspects => MapAspects.Altitude;
```

Figure 28: The step has no requirements before being ran, and it adds altitude to the map.

Another thing the step generators define is the map aspects they require and add.
The required map aspects tell the map generator what needs to be in place for this step generator to work, while the added map aspects tell it what this step generator adds to the map. This is used to prevent them from being run out of order and warn the user ahead of time instead of messing up the maps.

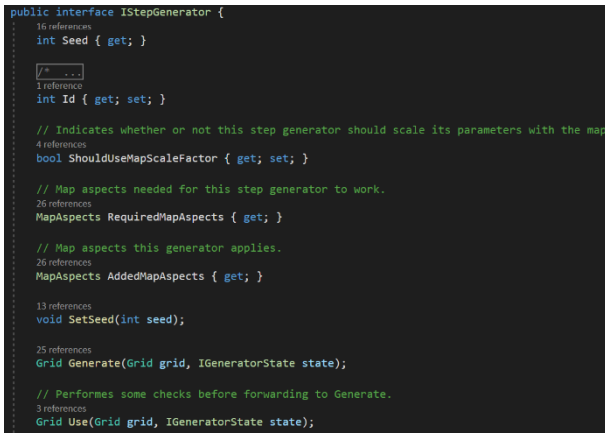


Figure 29: The interface for step generators.

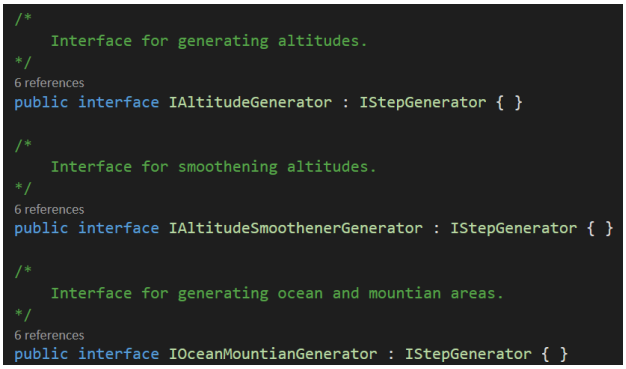


Figure 30: Some of the interfaces for specific steps.

All step generators implement the IStepGenerator interface so the rest of the code knows they are step generators. This requires them to define the map aspects they use, and the basic methods used to start the process. Other than that, they can add whatever configurations they want. Additionally, we have interfaces for the different kinds of steps we defined, but they are not required for most uses.
We also have a base class, which sets up a lot of the common things for step generators. This is only to make things easier, and only IStepGenerator is required to make it work.

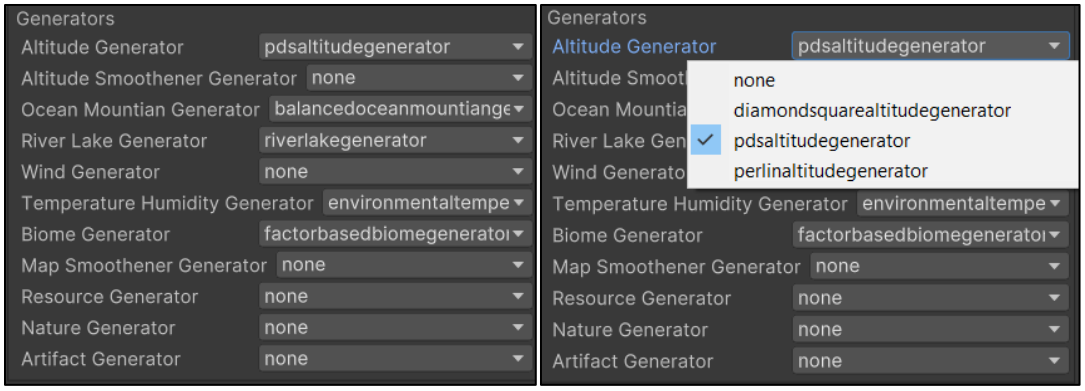


Figure 31: Drop down menus of different generators the program has available. (List is generated automatically)

In the testing GUI we designed, it will automatically find the step generators of different types and make lists of them by name for the user to pick between. For a step generator to show up here, they must do two things:

1) They need to implement the interface for one of the different types of steps.
2) They need to have an empty constructor (in addition to the non-empty one).

```
// Get all types branching off the root interface.
var types = assembly.GetTypes().Where(t => root.IsAssignableFrom(t)).ToList();

// Get all the interfaces for different step-generators.
var interfaces = types.Where(t => t.IsInterface && t != root).ToList();

// Get all concrete implementations of this interface.
var instances = types.Where(
    t => t.IsClass
    && !t.IsAbstract
    && !t.IsGenericTypeDefinition
    && t.GetConstructor(Type.EmptyTypes) != null
).ToList();
```

Figure 32: Code that creates a list of all step generator classes using reflection.

The lists are generated automatically using reflection at start-up. All classes that implement IStepGenerator are found. They are then grouped by the steps they implement. This way, if any new generators are created, they will automatically show up on this list without any extra work.

Altitude generator

This step generates the altitudes of the map. Each tile is assigned an altitude value between 0 and 1. This can be done with various noise algorithms, to give the terrain a desired look while keeping the terrain procedural.

Generator:	Required Map Aspects:	Added Map Aspects:	Configurations:
DiamondSquareAltitudeGenerator	None	Altitude	randomness
PerlinAltitudeGenerator	None	Altitude	scale
PDSAltitudeGenerator	None	Altitude	randomness, dsDepth

DiamondSquareAltitudeGenerator

This generator uses a Diamond Square algorithm to generate the altitudes. This normally requires the map to be a square, with each side being of length 2^n + 1. However, we can simply generate a larger square that fits this requirement and crop the map to fit any dimension. The randomness configuration determines how "grainy" the resulting map will look, with lower randomness resulting in more smooth and evenly spread altitudes.

PerlinAltitudeGenerator

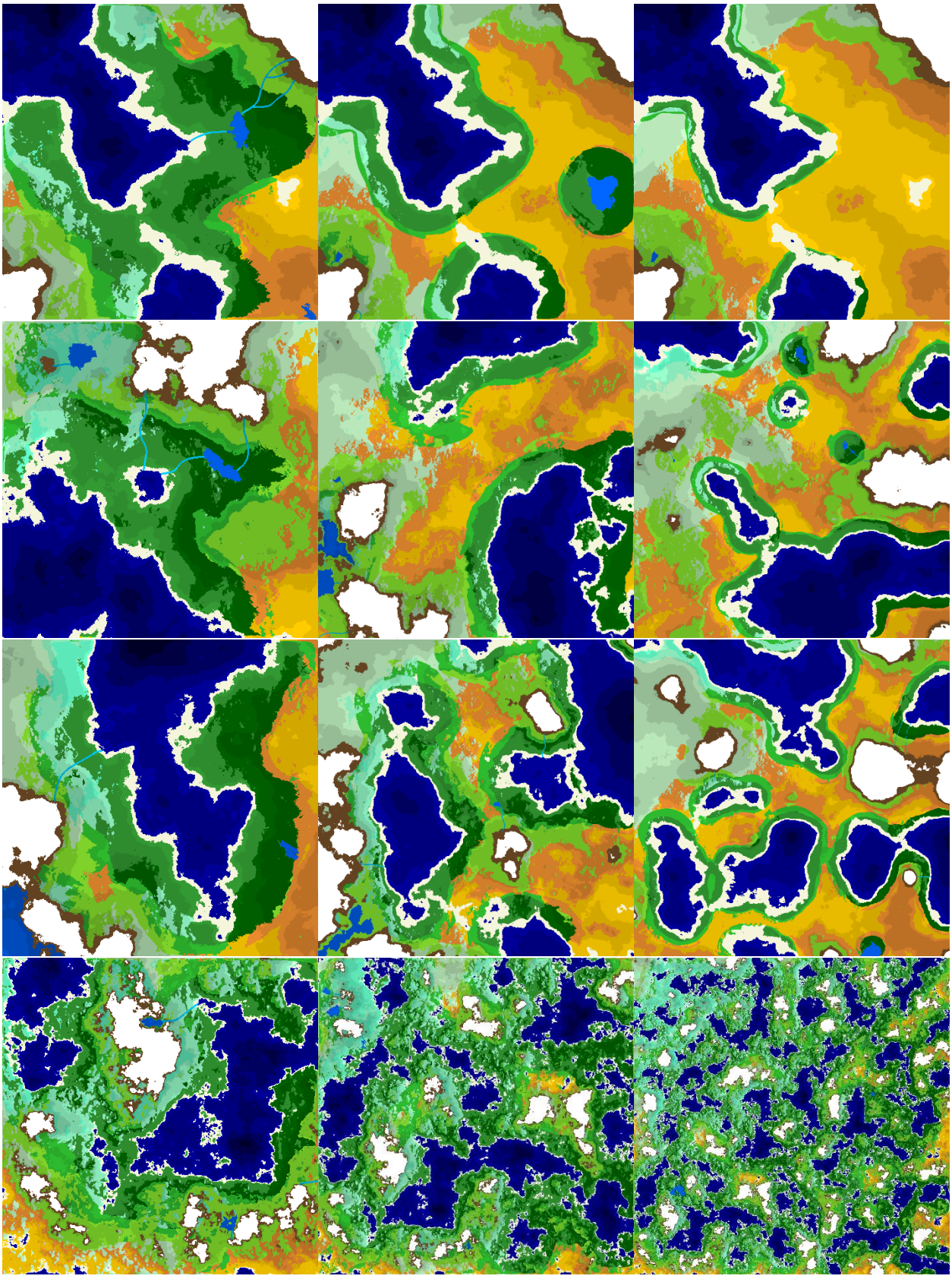
PerlinAltitudeGenerator uses Perlin noise to generate the altitude. Unlike Diamond Square, there is no requirement for the dimensions. Additionally, the Perlin function does not need to generate the entire map at the same time. With the right parameters, any one coordinate can be requested, and only that value is calculated. This makes it useful in infinitely generating worlds. However, since a requirement of our system is that the dimensions of the map be pre-defined, this would require some refactoring to take advantage of. The scale configuration affects how "zoomed out" the Perlin pattern appears.

PDSAltitudeGenerator

This generator uses a combination of Perlin and Diamond Square.

A benefit of Diamond Square is that it creates smoother looking environments. However, when the map size is increased, it will create the same map, but stretched out. This can create problems for larger maps as everything is further away, for instance causing the map to have less overall humidity and become less practical. Perlin solves this issue. If you make the map larger, Perlin will just continue adding more, effectively making the same map, but zoomed out. However, Perlin creates rougher terrain, which may not be a desired look.

PDS is a compromise between the two. Perlin noise is used to set the corner values of many smaller Diamond Square patterns. The size of these Diamond Square patterns is determined by the dsDepth configuration. This way the user can choose their preferred level of detail, while keeping the map repeating and uniform at large scales.



Images above: The same map generated using different altitude generators using default parameters.
From left to right: 513x513, 1025x1025, 2049x2049.
From top to bottom: Diamond Square, PDS_9, PDS_8, Perlin.

As can be seen above, the Diamond Square generator essentially stretches the same result out.
In this example, a few issues with scaling can be highlighted. Since the distance from everything grows, rivers fail to generate due to them being too long. Additionally, since humidity is generated from lakes, rivers, and oceans, the increased distance causes much dryer environments. Finally, the user might generally not want massive uniform continents, and instead prefer more varied landscapes. For Perlin, each map is just a zoomed out version of the previous, zooming from the bottom-left corner up and to the right. A big drawback to Perlin is that the terrain looks rough when looking at it close up.

For the maps generated with PDS, it gets a mix of the other two, achieving a similar smoothness to Diamond Square up close, but maintaining a similar scale of things as the map grows, like Perlin does.

Altitude smoothener generator

This step is used to smoothen the altitude map. This usually involves checking nearby tiles and adjusting the altitudes to reduce the difference and remove steep slopes.

Generator:	**Required Map Aspects:**	**Added Map Aspects:**	**Configurations:**
AltitudeAverageSmoothener	Altitude	None	List<(iterations, range, includeSelf)>
VariedAltitudeSmoothener	Altitude	None	iterations, range, minValue, randomness
ThermalErosionSmoothener	Altitude	None	List<(iterations, altDiffThreshold)>

AltitudeAverageSmoothener

This generator goes through each tile, checks its surrounding tiles, and sets the altitude to the average of its surroundings. This step can be repeated. This results in smoother terrain, especially if there is a sudden big difference between adjacent tiles. The configuration is a list of steps the generator goes through for each tile. For instance, it could do 2 iterations at range 3, where each tile includes itself in the calculation, the 1 iteration at range 2, not including itself.

VariedAltitudeSmoothener

This generator uses Diamond Square to pick how much each area gets smoothened. For each iteration, every tile is smoothened based on their noise value, before that value is reduced for that tile. When the noise value goes below a certain minimum, that tile no longer gets any smoothening applied. This allows the map to have some areas that are smoother than others, creating a more varied look.

ThermalErosionSmoothener

This generator simulates thermal erosion, by moving altitude downwards along steep slopes. For each tile, if a surrounding tile is far enough below it, a fraction of the altitude difference is transferred to the

lower tile. This smoothens the steepest parts of the map, without altering the texture of the more even sections. The configuration is a list of steps, each with the number of iterations and the altitude difference needed to start a transfer.

Ocean mountain generator

This step sets the ocean and mountain level, which are saved in the map's metadata. All tiles below the ocean level become ocean, and all tiles above the mountain level become mountain. Sand is always placed next to ocean as a buffer, and stone is similarly placed next to mountain. These are all special biomes that usually should not be overwritten. The altitude of an ocean tile represents the ocean floor, while the ocean level represents the surface.

Generator:	**Required Map Aspects:**	**Added Map Aspects:**	**Configurations:**
RawOceanMountianGenerator	Altitude	OceanMountain	waterCutOff, mountianCutOff
BalancedOceanMountianGenerator	Altitude	OceanMountain	desiredWater, desiredMountian, precision

RawOceanMountianGenerator

This generator sets the ocean and mountain level directly based on the configurations waterCutOff and mountianCutOff. This means that the fraction of the map that becomes ocean or mountain depends on how the altitude is spread out. For example, a map that has a sharp bell curve of altitude distributions will have few points that are very high or very low, and therefore less ocean and mountain, compared to a map with a flatter curve.

BalancedOceanMountianGenerator

This generator instead sets the ocean and mountain level based on the desired fraction of ocean and mountain. This means that the amount of ocean and mountain is not affected by the altitude distribution. This is done by fist partly sorting the tiles by altitude, resulting in several chunks, where all tiles in a given chunk have a lower altitude than the chunk above it. The cutoff points are set to the boundary between chunks that is closest to the desired ratio. The precision configuration determines the number of chunks, and therefore how close to the desired fractions the cut-offs will get.

River lake generator

This step generates rivers and/or lakes, which are stored as biomes on each tile. Each lake has their surface level saved in the metadata, as the tile's altitude represents the bottom of the lake.

Generator:	**Required Map Aspects:**	**Added Map Aspects:**	**Configurations:**
RiverLakeGenerator	Altitude, OceanMountain	RiverLake	riversFromStone, loneLakes, lakesFromRivers, riversFromLakes, numRiversWanted, numLakesWanted, sight

ClosestOceanRiverGenerator	Altitude, OceanMountain	RiverLake	adjustAltitude, numRiversWanted, minRiverLength, maxRiverLength, numCurves, curveSize

RiverLakeGenerator

This is the most complete solution we have for rivers and lakes. It can spawn rivers from stone, lone lakes, lakes from rivers, and rivers from lakes. The user can specify how many rivers they want, how many lakes, and the sight value used when rivers spread downwards.

Spawning rivers and lakes

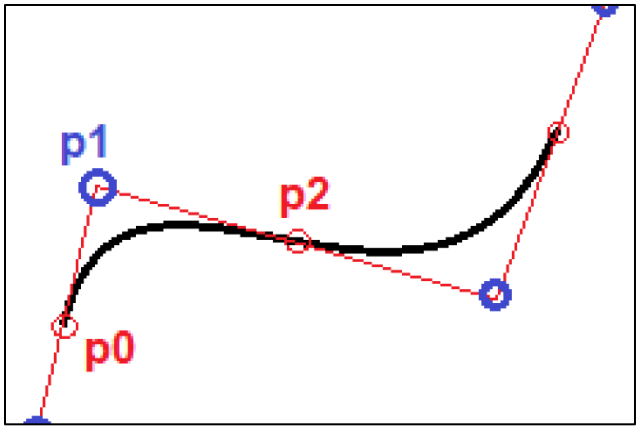
In RiverLakeGenerator, rivers can spawn from stone tiles at high altitudes, lakes can spawn in hollows by themselves, lakes can spawn from rivers flowing into hollows, and rivers can spawn from overflowing lakes.

When a river spawns from stone, it looks for a continuous downwards path to the ocean or a local minimum. Starting at a randomly chosen stone tile, the surrounding tiles are checked for the lowest altitude. If none are lower, then range it looks at is incremented, up to a maximum sight value. Once a lower tile is found, this pattern is repeated from there, adding each tile to a list until ocean is hit or a no lower tile can be found within sight. The river could also merge with another river, hit an existing lake, or go past the edge of the map. In all cases, the river is generated, provided it is not too short. If there is no lower tile within sight, the river has hit a local minimum, and a lake is spawned in the hollow, provided lakesFromRivers is turned on. The river generates after this lake has generated.

A lone lake spawns in a similar way, starting from a stone tile and searching for a local minimum, only the river does not generate. When any lake generates, it can overflow into a new river, provided riversFromLakes is turned on. This river spreads down in same way as a river spawned from stone.

Generating rivers

River generation uses the list of tiles found when searching downwards as a basis for Bezier curves. First, to prevent the river from having many tiny curves, and instead give it wider arcs, any points that are too close to each other are removed from the river list. Next, to ensure the Bezier curves connect to each other smoothly without any sudden changes in angle, a point is added between each existing point in the list, using the following pattern, and the resulting list is used to generate Bezier curves:



Blue circles represent the initial river points. Red circles represent the added points. These new points are placed at the coordinates between two consecutive river points, or the midpoint of the red lines. When making a Bezier curve, red points are always used for p0 and p2, while blue points are used for p1. The p2 point of a curve will be the p0 point of the next curve. This pattern ensures the Bezier curves always connect smoothly, with no sudden changes in angle, as the end of one curve and the start of the next are both at the angle of the red line.

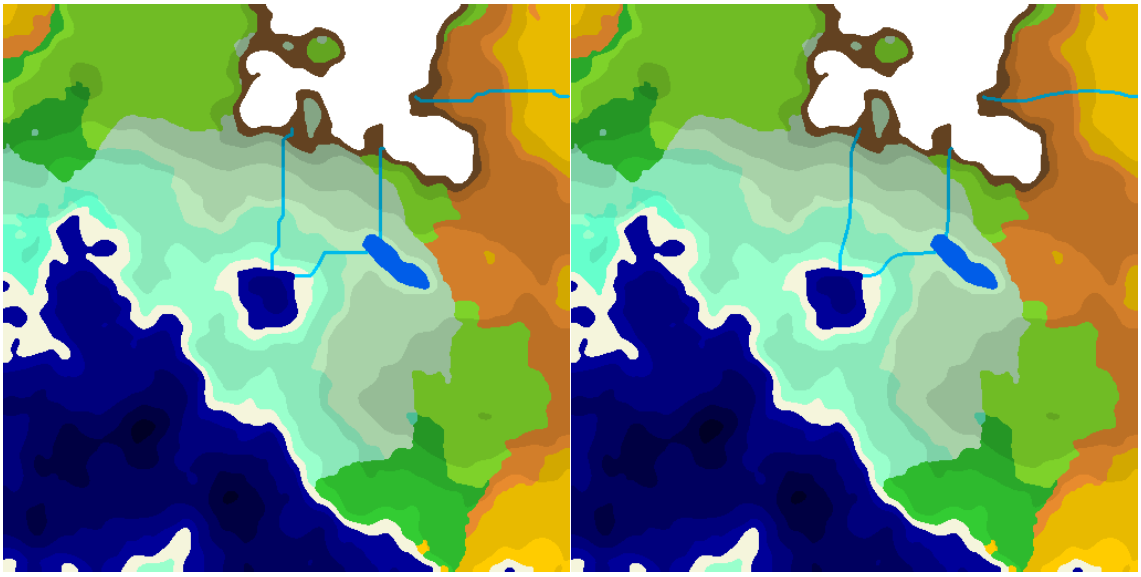


Figure 33: Same map without and with bezier curves.

All points along the curves within the river's width are then made into river biomes. In addition to changing the biome of the tiles, the altitude may be changed to prevent the river from ever flowing upwards. This is done by simply saving the lowest altitude so far, as the river generates downwards. If a tile is ever higher than this, it is instead set to the saved value.

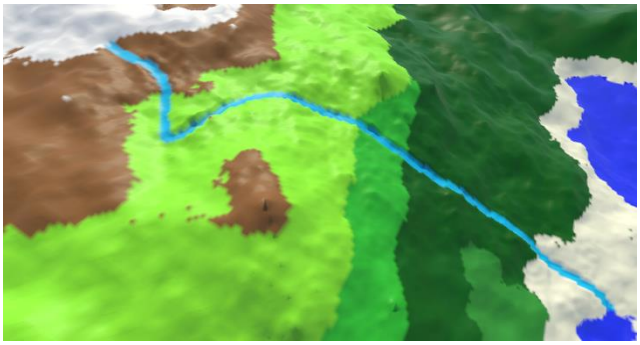


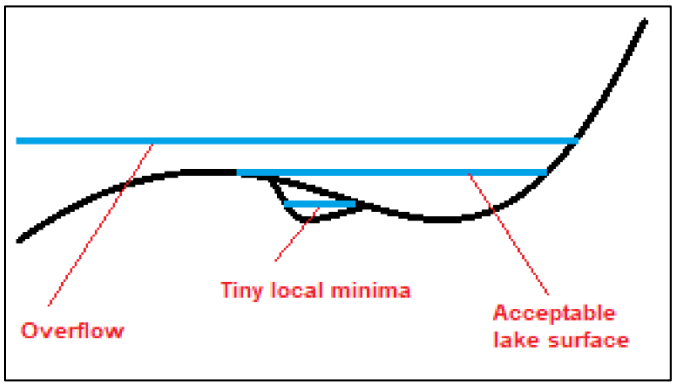
Figure 34: This river is carving slightly into the ground, as the terrain is rough and would otherwise result in the river sometimes flowing upwards.

Generating lakes

Lake generation is done using a set of valid lakes tiles. Starting from the local minima found by the river, adjacent tiles are added to the set, provided they are below a certain altitude limit, as if pouring water into the hollow. When no more adjacent tiles are below the altitude limit, the limit is slightly increased, as if the water surface is rising, and the process repeats. Once the size of the set gets too big, the changes of the last altitude limit increment are reverted, so that all tiles in the set are below the highest valid altitude limit. All tiles in the set are then made into a lake biome, and the highest tile's altitude is saved as the lake's surface level.

The reason for this method is to deal with the "overflow" problem. Since the lake starts in a local minima, adding tiles up to an altitude limit works fine, until it escapes the hollow and flows downwards. Therefore, we have to revert the altitude increase that caused this overflow.

We could simply stop the moment the terrain starts going downhill, as this means the local minima has been escaped, but this causes another problem. With uneven terrain, larger hollows will often contain smaller hollows. If a river ends in one of these smaller hollows, we want it to spread to the larger hollow, otherwise we would only have tiny lakes. What constitutes "overflow" is somewhat subjective, as you could argue that even if a lake spreads to cover more than half the map, it is still contained within a massive hollow. Therefore, overflow is determined by a lake size limit.



When spawning rivers from lakes, we use the set of all tiles on the edge of the lake, which is found as part of the lake generating process. Once the lake is generated, the lowest non-water tile within a

certain range of all the edge tiles is found. The edge tile closest to this point becomes the starting point of the river, and the lowest tile becomes the second point, as the river spreads further down. If no tile is found that is lower than the edge, a river cannot generate from the lake.

ClosestOceanRiverGenerator

In this generator, rivers are generated in a much simpler way. A random stone tile is chosen, and its surroundings are searched for the closest ocean tile. Then, a Bezier curve is made between the two points. The user can specify how many curves they want each river to be made of, and how curvy these should be. The river generation fails if this curve collides with another river, as this would create strange patterns.

Wind generator

This step generates a windmap. Each tile has a wind direction and magnitude stored. Wind could be used to affect other factors, like humidity and temperature, and ultimately determine biomes. Another application could be to simulate erosion and affect the heightmap, or to simply use the wind as a mechanic in the game the map is used in.

Generator:	**Required Map Aspects:**	**Added Map Aspects:**	**Configurations:**
LocalTilesWindGenerator	Altitude	Wind	sight, altitudeSensitivity
FlowingWindGenerator	Altitude	Wind	altitudeSensitivity

LocalTileWindGenerator

In this generator, each tile is given a wind magnitude and direction based only on its surrounding altitudes. Wind generally flows from west to east, with variations in angle based on the vertical tilt around the tile. If the average altitude north of the tile is greater than the south, it is tilting southwards, and the wind angle will also be tilted southwards, and vice versa. The greater the tilt, the greater the change in angle. The magnitude is based on the horizontal tilt. If the total altitude to the east of the tile is greater than the west, it has a forward-facing tilt, and the magnitude is increased, and vice versa. For oceans and lakes, the surface altitude is used, rather than the bottom, as wind flows along the water surface.

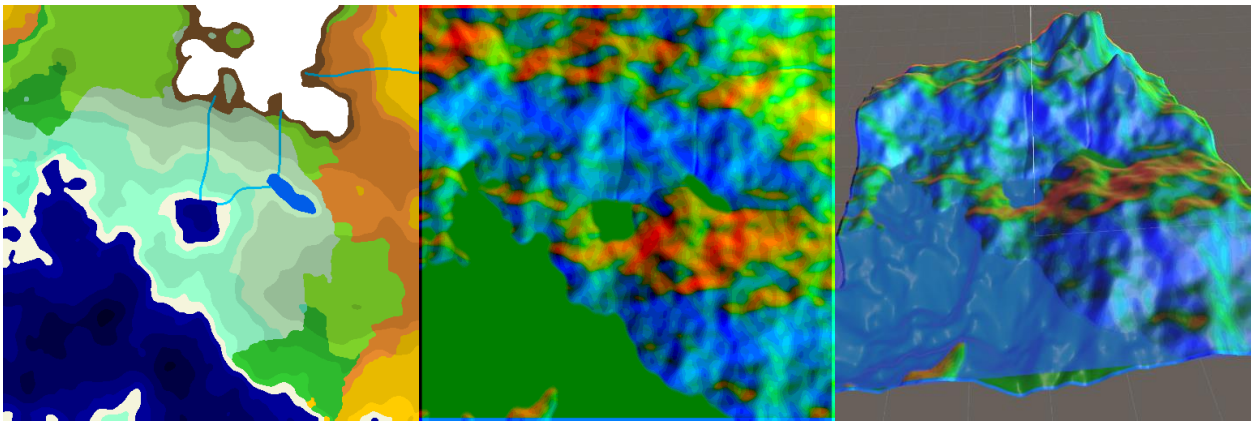


Figure 35: A map along with its wind map and the wind map in 3D.

FlowingWindGenerator

In this generator, wind can by contrast carry over as it flows from tile to tile, from left to right. This results in more "flowing" streams of wind that can remain or dissipate over greater distances. For each column, each tile spreads its wind to the 3 tiles in front of it. The tile directly in front of the source tile has a straight forward-facing angle, while the two diagonals have a roughly 90-degree angle, pointing away from the source tile. Each source tile has a certain total magnitude to give away to the 3 tiles ahead. The ratio of that magnitude that each tile receives is based on the altitude factors and the direction factors.

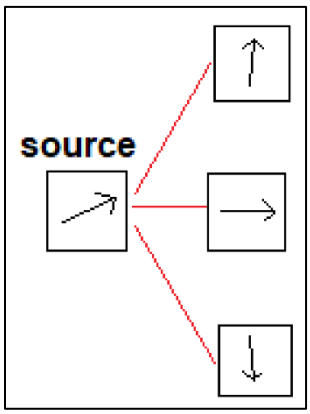


Figure 36: A source tile adds wind vectors to 3 tiles ahead. The angle of the vectors added is always the same, but magnitude varies based on altitude and wind direction.

The altitude factors are determined by the altitude difference between the source tile and the receiving tile. The lower the receiving tile is relative to the source tile, the greater the factor. The direction factors are determined by the direction of the wind in the source tile. The more the wind angle tilts up, the greater the factor for the upwards tile, and vice versa. If the wind in the source tile flows straight ahead, each of the 3 tiles has the same direction factors. This ensures there is always some dispersion of wind. The two factors are then both normalized so the sum of the 3 values is 1, before being added together and normalized again. These 3 values are the final ratio of the magnitude that each of the 3 tiles receive from this source tile. As tiles in columns ahead receive wind vectors from multiple source tiles, their

vectors are added up, and they get their final direction and magnitude. Once every tile's wind is set, the wind magnitude is normalized based on the highest value, so that all magnitudes are between 0 and 1.

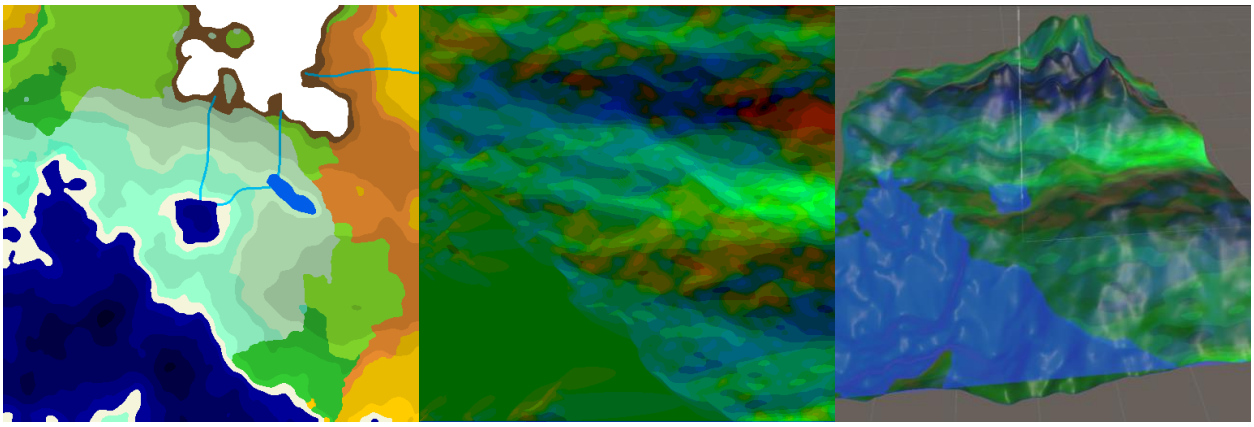


Figure 37: A map along with its wind map and the wind map in 3D.

Temperature humidity generator

This step generates factors like humidity and temperature. These can be used to determine the biomes.

Generator:	**Required Map Aspects:**	**Added Map Aspects:**	**Configurations:**
DiamondSquareTemperatureHumidityGenerator	None	TemperatureHumidity	tRandomness, hRandomness
PerlinTemperatureHumidityGenerator	None	TemperatureHumidity	scale
EnvironmentalTemperatureHumidityGenerator	Altitude, OceanMountain, (can use Wind)	TemperatureHumidity	humiditySpreadDistance, chunkSize, dispersionSteps, dispersionRange, dispersionProp, useWind, windDispersionTake, windDispersionGive

PerlinTemperatureHumidityGenerator

This generator creates humidity and temperature using Perlin. Each factor is determined by a separate noise map, resulting in two overlapping maps.

DiamondSquareTemperatureHumidityGenerator

This generator creates humidity and temperature using Diamond Squared. Each factor is determined by a separate noise map, resulting in two overlapping maps.

EnvironmentalTemperatureHumidityGenerator

This generator is meant to simulate a natural environment determining humidity and temperature. It first spreads humidity from water sources, then uses wind to spread it, if enabled. Temperature is then

generated from noise, followed by a sunlight simulation and some dispersion. Finally, humidity makes the temperatures less extreme, and temperatures at high altitudes are reduced.

Humidity

Chunking

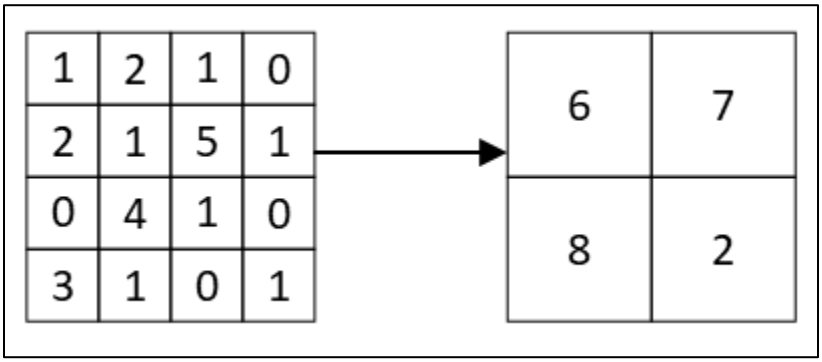


Figure 38: A 4x4 grid being converted into 2x2 chunks. Each chunk has the sum of all the tiles it represents.

The first thing we do is group the tiles into chunks. This is only done for the first step because of the heavy performance cost it has. By grouping the tiles together into chunks, we can drastically reduce the number of steps we have to perform. The downside is that the result becomes blocky, which we address shortly after.

This chunking cuts the time needed substantially. For example, on a 1025x1025 map with humidity spread range of 120, running with no chunks took over 13 minutes, while 4x4 chunks took 12 seconds.

The reason for this performance benefit is that chunking of size N cuts the number of tiles to spread from by N^2 and the distance to spread each tile for by N.
As an example, for a map of 1024x1024 tiles, with spread range of 160, and chunk size of 4, there would be 256x256 chunks, each of which only has to spread 40 chunks.

Spread

The first step we perform in this process is to spread humidity from the different sources on the map.

```
{ BiomeType.Ocean,    0.7f },
{ BiomeType.River,    1.3f },
{ BiomeType.Lake,     0.7f },
{ BiomeType.Glacier,  0.7f },
```

These are biomes like ocean, river and lake. Each source has a strength, which defines how much humidity it spreads. This can be tweaked. When chunked, the sources of all tiles in the chunk are added together.

For each chunk with humidity sources, we spread humidity to all tiles in an area around them, with less humidity spreading to tiles further away. This simulates humidity from these sources spreading to their nearby areas.

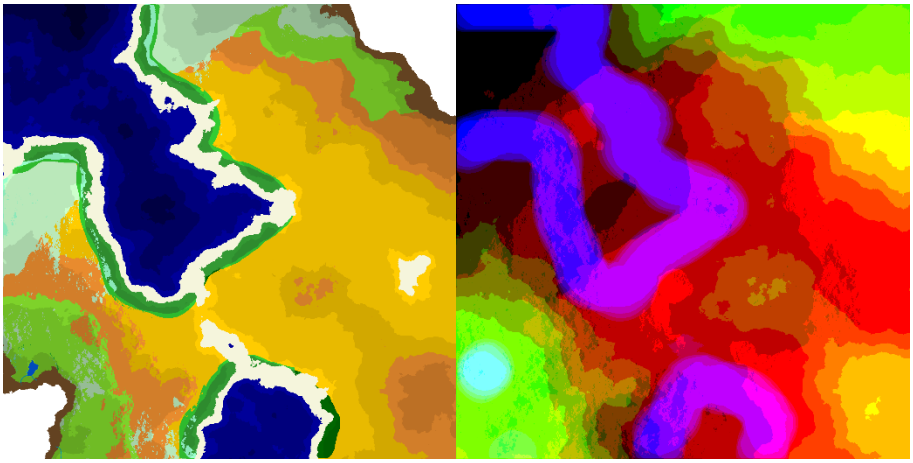


Figure 39: Factor map showing that humidity only spreads from the coast of the ocean.

Another detail is that chunks with no land nearby will not spread humidity. There are two benefits to this:

1) We avoid costly spreading for chunks that will not benefit from it. This check uses a shorter range, and checks in a diamond pattern, which is much cheaper.
2) The coastal tiles will not have as much more humidity compared to other tiles, which causes issues when normalizing the humidities.

Unchunking

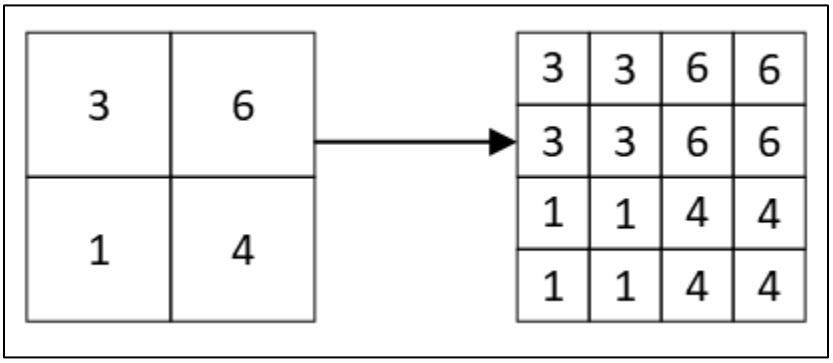


Figure 40: A grid of 2x2 chunks being split back into tiles.

After spreading the humidity, we convert the map back to tiles. Here, all tiles in a chunk gets the full humidity of the chunk.

Dispersion

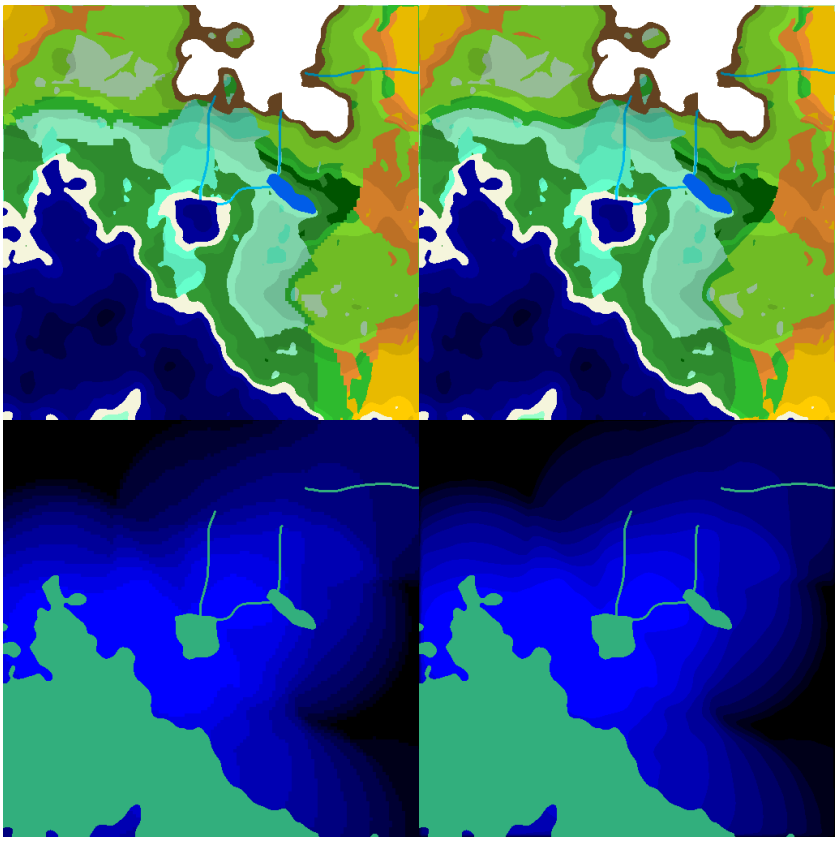


Figure 41: Side-by-side of humidity of a map, with and without dispersion. (Size: 513x513, chunk size 4)

As a result of spreading humidity in chunks, we get a blocky look. To address this, we add another step to the process: dispersion.

In this step, each tile will take a portion of its humidity and give it to the tiles around it. When dividing the humidity on its neighbors, we divide it by who has the least humidity. That way, the humidity spreads outwards, creating a smoother look with relatively little cost to performance.

Wind

The next step is to have the wind carry humidity across the map. This helps reduce the circular look of the humidity caused by it spreading in a circle and make it more irregular. It also simulates how wind transports humidity in reality, like how wind can carry rain from the ocean and make an area more wet, especially if the wind hits a mountain and cannot go further.

```
for (var x = 0; x < w - 1; x++) for (var y = 0; y < h; y++) {
        var windStrength = grid[x, y].windMagnitude;
        var humidity = hGrid[x, y];
        hGrid[x, y] -= humidity * windStrength * windDispersionTake;
        hGrid[x + 1, y] += humidity * windStrength * windDispersionGive;
    }
return hGrid;
```

Figure 42: The code responsible for wind moving humidity.

The way this is done is to take a certain percent of a tile's humidity and give a certain percent to its neighbor based on wind strength. How much of a tile's humidity is taken and how much is given to the

next tile are separate values, so more humidity can be passed along than taken, or vice versa. If no humidity should be gained or lost, the same number can be used for both. These numbers are then multiplied by the wind strength on the tile, so if a tile has 50% wind strength, only half the specified amounts will be taken and given. This way, more windy areas have more humidity moved.

Using the wind to transport humidity is optional. If enabled, the step generator requires wind to be added to the map, otherwise it does not require it.

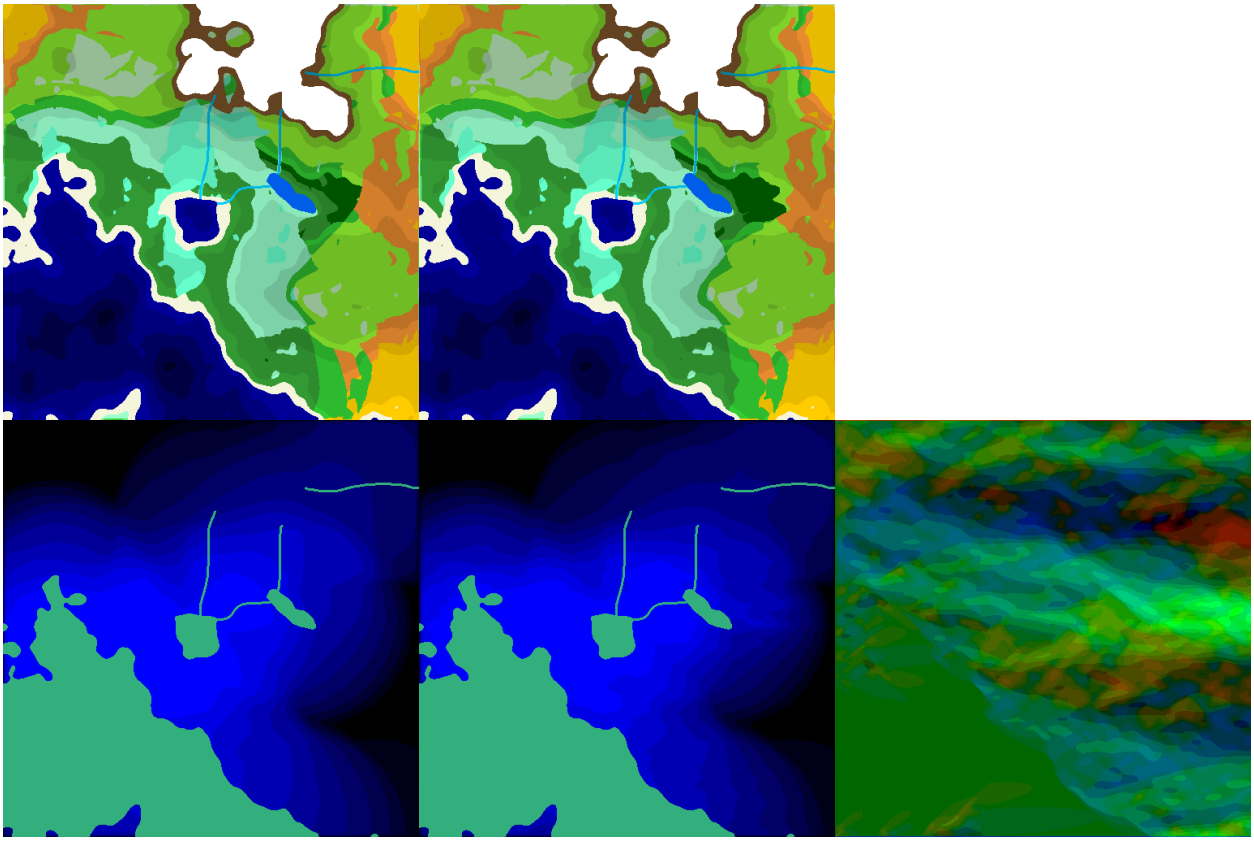
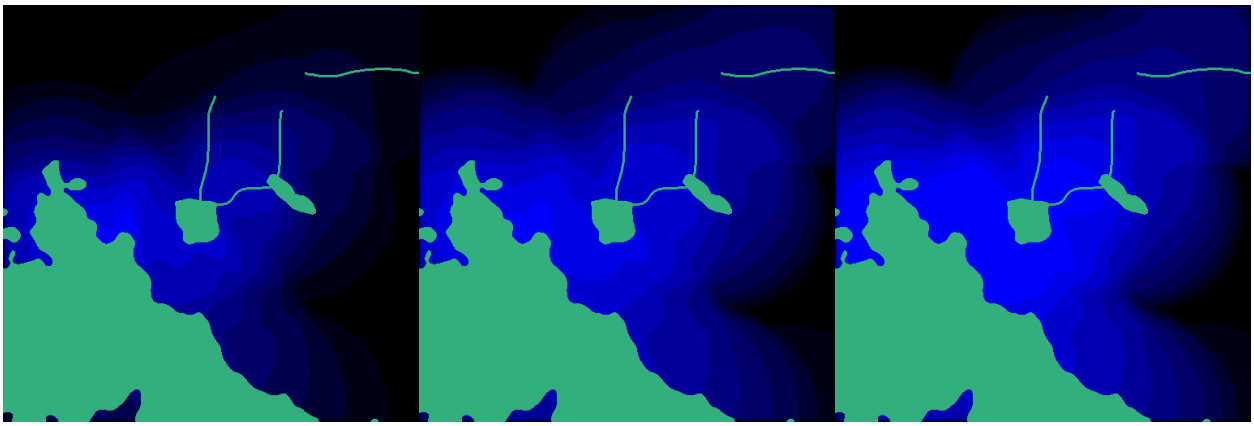
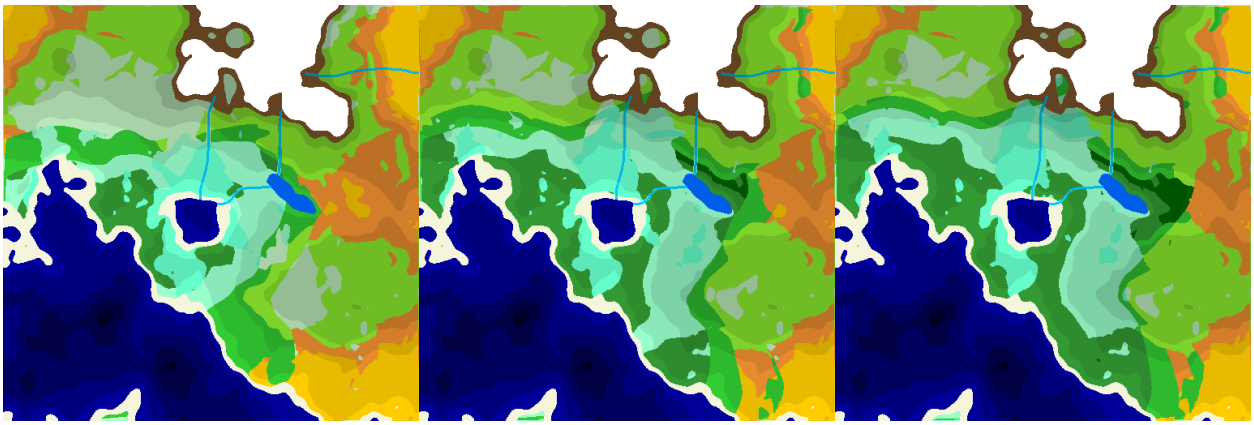


Figure 43: Before and after adding wind. (Last image is the wind map). The strong wind on the right carries humidity from the lake eastwards.

Normalization





Above are the same map generated with different normalization rules. From left to right:

1) Just using the tile with the highest humidity that is not also a humidity source.
2) Same as 1, except using square root on all tiles to reduce the difference.
3) Same as 2, except the cap is between the max and an average.

After humidity has been generated, we need to normalize it, so it fits within a range of [0, 1].
A common problem we would run into was that some places, especially the coast, had a lot more humidity than the tiles further out, so when normalized, there was barely any humidity. One thing we did to help reduce the difference was to get the square root of the humidity for each tile. That way, big numbers like 100 turned into 10, with numbers like 12 turning into 4.

While this helped, it did not give us the kind of results we wanted. We realized the problem was that a few tiles had way more humidity than others, causing the max to be way too high. To address this, we changed the upper limit from the highest value to something a little more complex. We take the average humidity, then take the average humidity of tiles with more than the average. We then found the average between that and the maximum. This reduces the effect of tiles with absurd humidities on the overall result and gives a more natural looking spread.

Temperature

Random noise

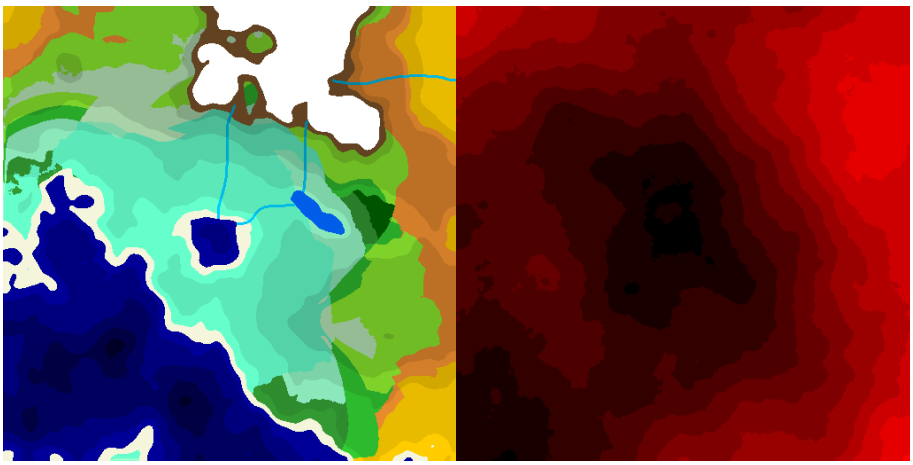


Figure 44: Temperatures with only noise.

First step for temperature is to create a noise map using Diamond Square. This gives us something to start from and gives the map a degree of unpredictability.

Sunlight sim

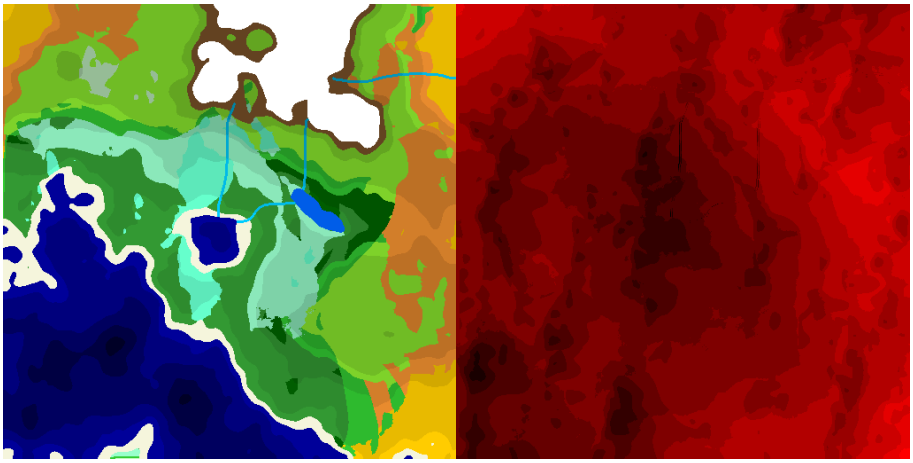


Figure 45: Same map with sunlight simulation added.

Next up, we simulate sunlight on the map. We came to a relatively simple solution. We go row by row, looking at the altitudes of each tile and casting a shadow at a certain angle. For each tile we move, the shadow goes down by a certain amount. If the tile is above the shadow, it gets heat from the sun, and the height of the shadow gets set to its height. If the tile is below the shadow, it gets no sunlight and casts no shadow.

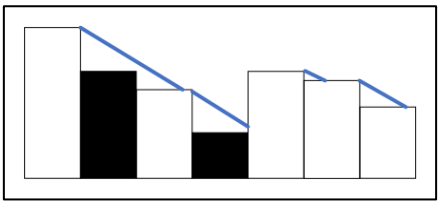


Figure 46: Sunlight simulation. The blue lines represent the border of the sunlight. The rectangles represent tiles, filled rectangles are tiles that didn't get hit by the light.

Further, to simulate different times of the day, we repeat the process above with different angles, simulating the sun being higher or lower in the sky. This way, the strength is different depending on how sharp the drop is.

Minor dispersion

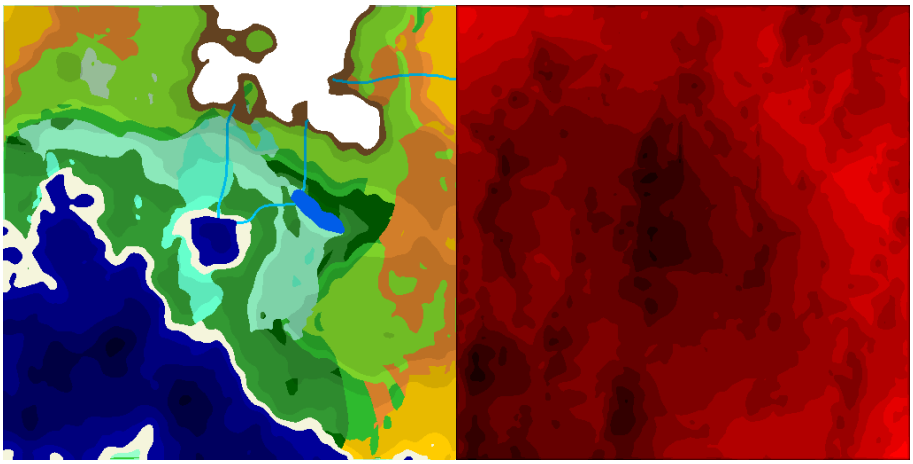


Figure 47: Same map, with temperature dispersion added.

This process left some holes where the temperature is much lower than its neighbors. To address this, we added a small amount of dispersion using the same algorithm we used to disperse humidity.

Normalization

After that, we normalize the temperature to be within [0, 1]. Unlike humidity, temperature does not vary as much, so we used the highest temperature as maximum.

Altitude effect

```
// Altitude reduces temperature slightly.
t *= 1f - grid[x, y].altitude.Lerp(altMid, altMax) * ALTITUDE_TEMP_MODIFIER;
```

We then reduce the temperature slightly based on altitude. This helps simulate how mountains usually get colder. We do this by first finding the midpoint between ocean and mountain. Any tile above that gets a temperature reduction proportional to how far up it is. At the highest, it is reduced by 20%.

Humidity effect

```
// Humidity pulls temperature closer to the middle.
t = (
    (t * 2 - 1)
    * (1f - HUMIDITY_TEMP_MODIFIER * hGrid[x, y].Lerp(MIN_HUMIDITY_FOR_TEMP_MOD, 1.0f))
    + 1
) / 2;
```

Lastly, the temperature is pulled closer to the center depending on how humid it is. This is to simulate how humid areas usually have fewer extreme temperatures. If the tile has a humidity of more than 0.4, it will have its temperature pulled closer to 0.5 the more humid it is, up to a max of 20% at 1.0 humidity.

The formula for this is: ((t*2 – 1) * mod + 1) / 2, with t meaning temperature, and mod meaning the modifier in range [0.8, 1.0].

Clamp

```
tGrid[x, y] = t.Clamp(0f, 1f);
```

Lastly, we clamp the temperature to be between [0, 1].

Rationale

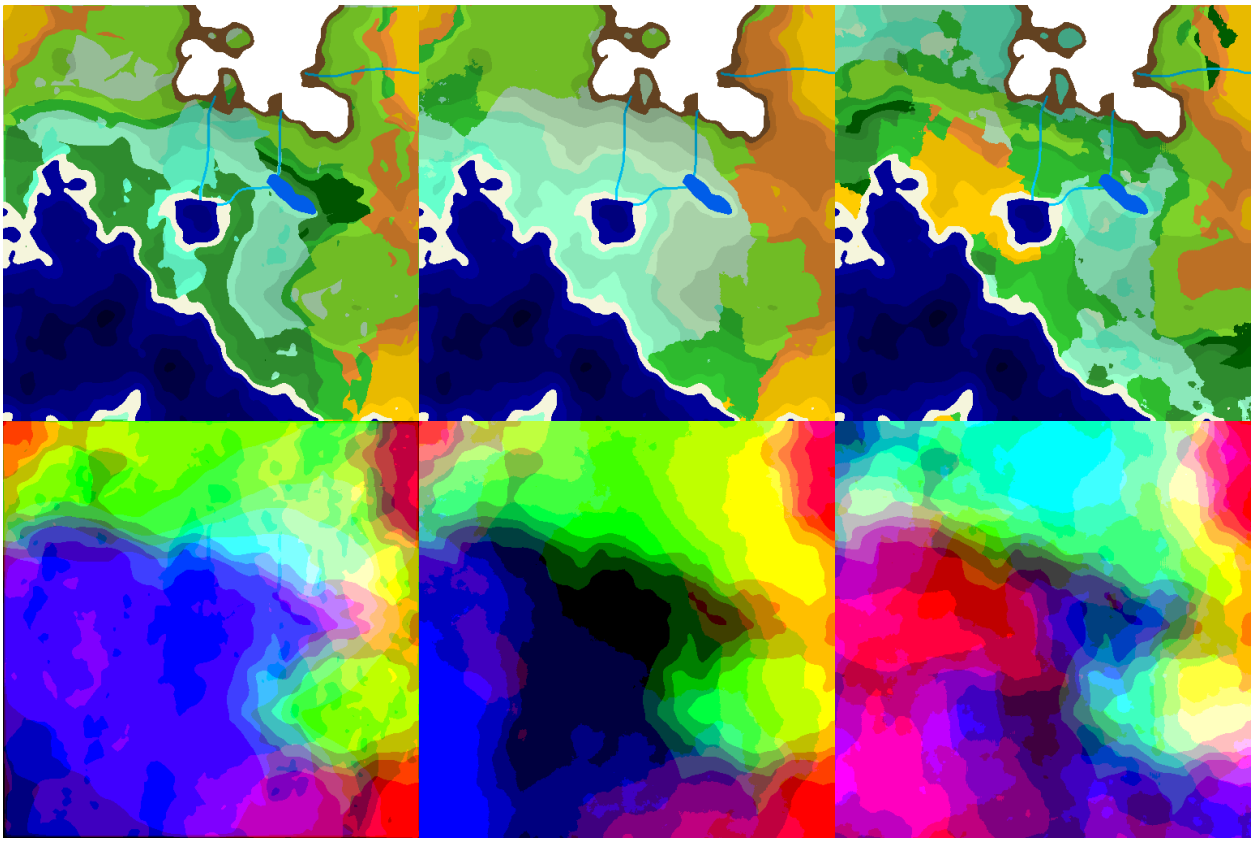


Figure 48: The same map with different generators temperature and humidity. Left-to-right: Environmental, Diamond Square, Perlin.

The reasoning behind this model is to simulate multiple different factors that affect temperature and humidity, with a small amount of noise for unpredictability.

By having many different factors, each doing a small amount, we create a much more complex and interesting map than having a single factor decide everything.

Biome generator

This step sets the biomes of the map. This could be done entirely with noise, based on altitude, factors like temperature and humidity, or any combination of these. Each biome has information about it stored in the metadata of the map, such as number of tiles.

Generator:	**Required Map Aspects:**	**Added Map Aspects:**	**Configurations:**
OneDNoiseBiomeGenerator	None	Biomes	biomeCutoffA, biomeCutoffB
AltitudeBasedBiomeGenerator	Altitude	Biomes	biomeCutoffA, biomeCutoffB, biomeCutoffC, biomeCutoffD, biomeCutoffE

ExpandingBiomeGenerator	Altitude, OceanMountain	Biomes	None
FactorBasedBiomeGenerator	Altitude, TemperatureHumidity	Biomes	biomeCutoffA, biomeCutoffB, biomeCutoffC, biomeCutoffD

OneDNoiseBiomeGenerator

This generator uses Perlin noise to determine biome. For each tile, the noise value is used to determine an index, which is used in the biome map to get the correct biome. The cutoffs between indexes can be chosen by the user.

AltitudeBasedBiomeGenerator

This generator uses altitude to determine biome. For each tile, the altitude value is used to get the index of the biome map. The cutoffs can be determined by the user.

ExpandingBiomeGenerator

This is a generator we created early on. It spawns seeds for different biomes across the map, then spreads them around slowly.

FactorBasedBiomeGenerator

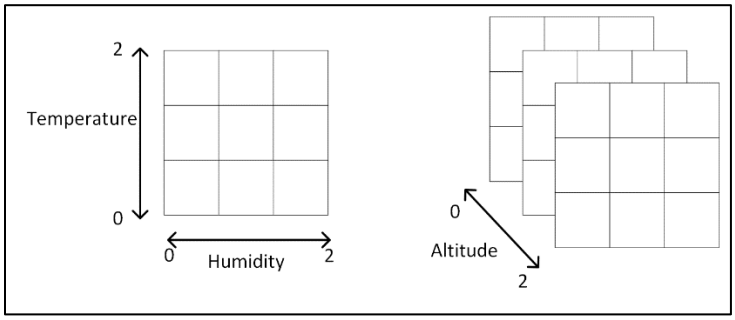


Figure 49: Diagram showing how the factors map onto the matrix. (3x3x3 model for illustration)

Using 3 factors that have been generated: altitude, humidity and temperature, this generator picks a biome for each tile. Each factor is converted from [0, 1] into a number between 0 and 4. This number is then used to look up a 3D matrix, where each axis corresponding to a factor, to find a biome for the tile.

This generator bases itself on the factors being distributed in a good way. If the factors are too uniform, the biomes become large. If the factors are too chaotic, the biomes look random. This means that this step generator can piggyback off the step generators for altitude, humidity and temperature. It also makes this an excellent way to try out step generators for the other factors and see what works.

The generator allows the user to send in the thresholds that say if a factor is very low (0), low (1), normal (2), high (3) or very high (4). It also allows the user to send in a 5x5x5 matrix used to pick biomes. This allows the user to decide what biomes to use. For example, an arctic map could use cold biomes

even when temperature is high. The matrix can use the same biome in different positions. That way, multiple different places can mean the same thing. This can be used to create more varied borders between biomes.

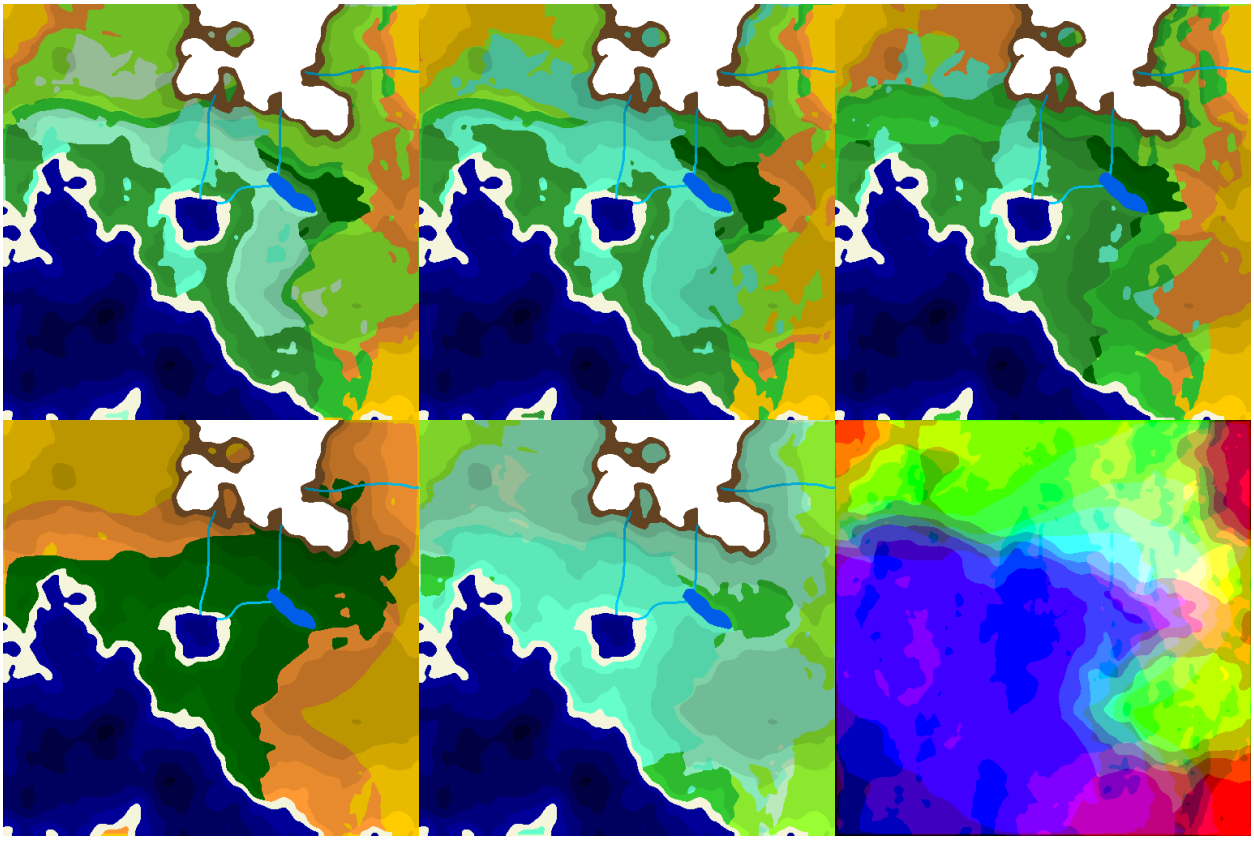


Figure 50: The same map generated using 5 different biome maps: Default, Simplified, Simplified2, Tropic, Nordic. (The 6th image is the factor map)

As can be seen above, this step generator allows the users to pick how they want to represent their map. They can control how factors map to biomes, how many biomes they want, and what how much certain biomes should border each other. They could for example have a lot of different biomes to make things more interesting or they could restrict the number of biomes to make it less chaotic.

Using this biome matrix, it is also possible to set themes for the map. For example, an arctic map could have no deserts and have cold biomes show up for higher temperatures, while a desert map could remove all cold biomes and have forests only show up at extremely high humidity and temperatures. Developers can easily create their own biome maps to decide how the factors map to biomes.

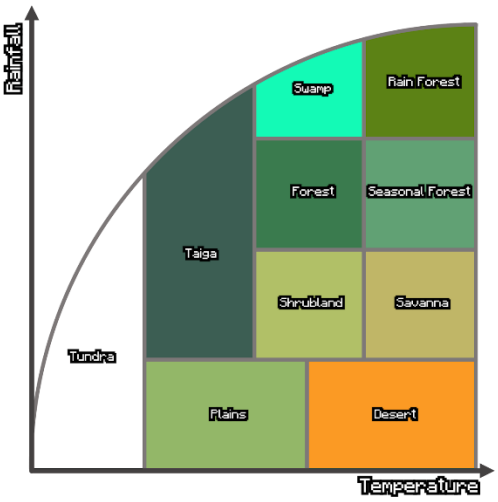


Figure 51: A similar system is used in Minecraft, where biome is based on rainfall and temperature, among other factors. This is a simplified graph. Taken from: https://www.alanzucconi.com/2022/06/05/minecraft-world-generation/

This algorithm is heavily inspired by the way Minecraft picks biomes based on the different factors each chunk has.

Originally, we built this with a 3x3x3 matrix. While this worked early on, we quickly noticed that the borders were clear and obvious. Later, we changed it to a 5x5x5 version and added more biomes. This increased the number of possible biomes from 28 to 125. With many more biomes, it became a bit too chaotic. In the end we ended up with a 5x5x5 matrix, but with the number of different biomes somewhere between the two previous versions. We instead used the extra positions to make more interesting borders.

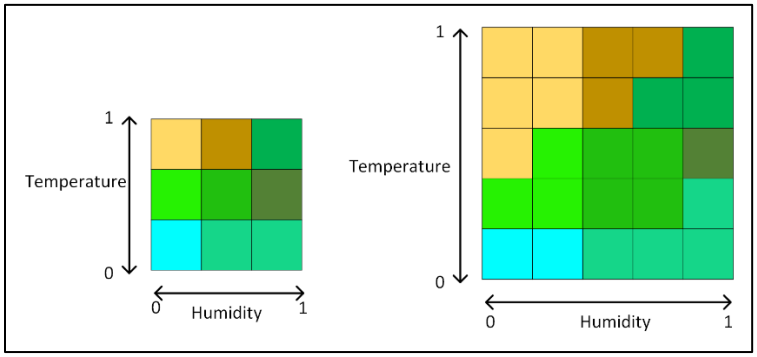


Figure 52: An example of a 3x3 matrix and a 5x5 matrix.

In the diagram above we see a 3x3 matrix and a 5x5 matrix. These are simplified versions for illustrative purposes, as they ignore the altitude aspect. The 5x5 one has the same biomes, but it has them spread in a less symmetric way, making it less obvious where factors change on the map.

Map smoothener generator

This step smoothens the map after biomes have been generated. This could mean tweaking the altitude of the map using biome information, or alternatively smoothening the borders between biomes.

Generator:	**Required Map Aspects:**	**Added Map Aspects:**	**Configurations:**
BiomeBasedAltitudeSmoothener	Biomes, Altitude	Altitude	BiomePars
CellularAutomataSmoothener	Biomes, Altitude	Altitude	iterations, range

BiomeBasedAltitudeSmoothener

This generator applies altitude smoothening to tiles using the same method as AltitudeAverageSmoothener, but can use different parameters and number of iterations for each tile depending on its biome. For instance, if the user wants deserts to have a smooth texture, it can be assigned more iterations of smoothening than other biomes, and greater range.

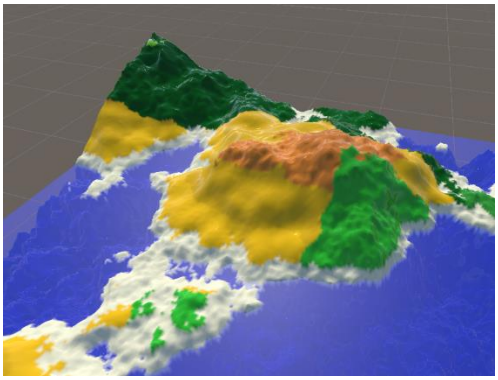


Figure 53: A map with biome-based altitude-smoothening. The yellow deserts have significant smoothening to imitate dunes.

CellularAutomataSmoothener

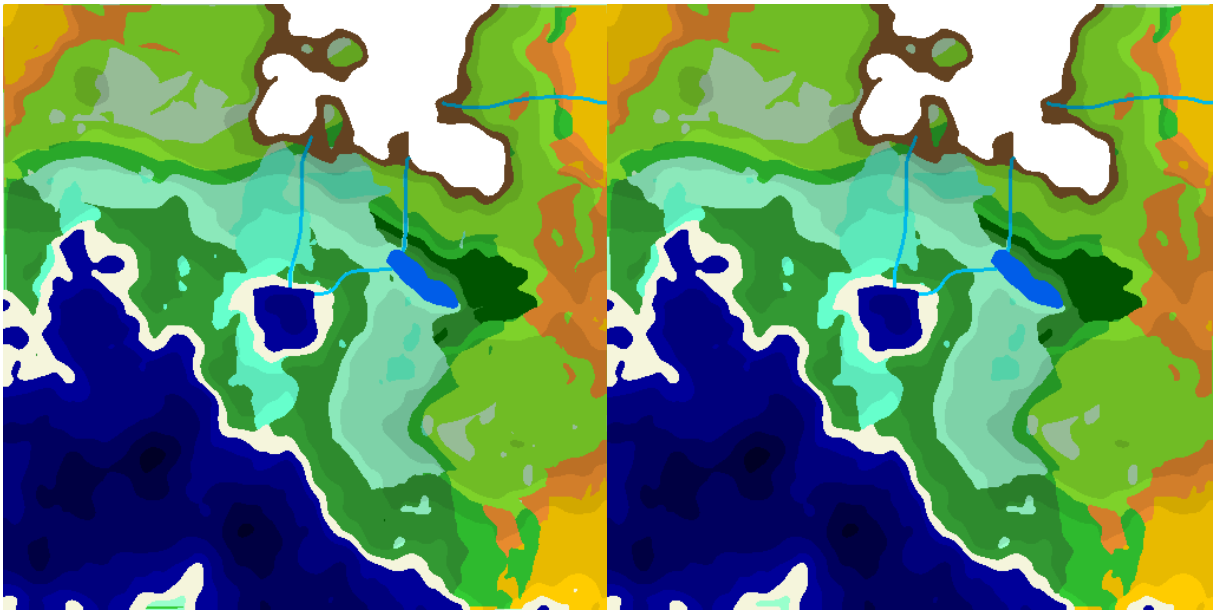


Figure 54: A map without and with CellularAutomataSmoothener. (2 iterations of range 2)

This smoothener looks at tile biomes and removes rogue tiles. It does this by going through the grid and for each tile, it looks at its neighbors in a diamond pattern. It counts the number of tiles in range of each biome type, with less weight the further away it is. If it is surrounded by more biomes of another type than its own type, it converts to that biome type. It takes two configurations:

- **Iterations** is the number of times it should repeat the process.
- **Range** is the distance it looks at neighbors.

Resource generator

This step generates resources. These are collections of tiles that can represent a resource in a game, like ore or berries, or something more abstract, like a denser section of a forest. They could also be thought of as sub-biomes, though they are stored independently of the biomes. A resource in a tile can also have a density value, which could be used for various gamplay mechanics, such as how much of a resource is available on each tile.

Generator:	**Required Map Aspects:**	**Added Map Aspects:**	**Configurations:**
PrimitiveResourceGenerator	Biomes	Resources	ResourceDist
FloodFillResourceGenerator	Biomes	Resources	resourceOverlap, escapeBiome, ResourceDist

PrimitiveResourceGenerator

This generator is the most simple resource generator. For each resource type, it places resources on random tiles of the valid biome types, until enough of that type has been placed. The density value is the same for each resource.

FloodFillResourceGenerator

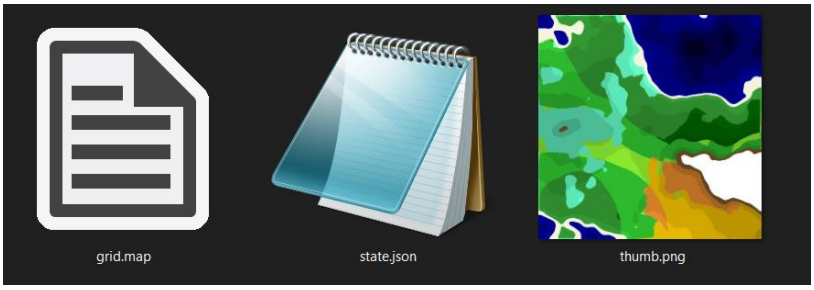
This generator uses a lazy flood-fill algorithm to spread resources from seeds. Each resource type can have a specified number of seeds spawned, biomes in which it can spawn, and decay rate. The decay rate is used to determine how far the resource spreads.You can also choose whether a resource should be able to spread outside its biome of origin, as well as how many different resources can be on a single tile.

Nature generator and artefact generator

These steps have no current generators. They are meant as inspiration for users for possible other steps to expand on, and to show how little code is required to add functionality. Nature generator could perhaps be a way to place details within biomes, like trees in forests or cacti in deserts. Artefact generator could place more complex structures like ruins or dungeons, with a predefined tailor-made architecture.

File system

Overall structure



When the map is saved, it is stored as multiple files in a folder. This is because they hold different information that is better stored separately.

Each map is given a folder with the same name as the map to make the files of the map easy to manage. Within the folder, there are 3 files.

- State.json — Stores overall information about the map itself.
- Grid.map — Stores information about each tile.
- Thumb.png — An image that gives users an idea what the map looks like.

State

This stores various pieces of information about the map itself as well as information needed to load the grid from file, such as width, height and how the tiles are encoded. For these reasons, the state file is loaded first.

```
{
  "Version": "v0.3",
  "Name": "Test1",
  "OceonLevel": 0.26,
  "MountianLevel": 0.88,
  "Width": 513,
  "Height": 513,
  "Seed": 9999,
  "GridImgCount": 0,
  "TileStorageConfig": {
    "Biome": 8,
    "Altitude": 9,
    "Temperature": 9,
    "Humidity": 9
  },
  "MapScalingFactor": 1.00391006,
  "MapAspects": "Altitude, Factors, Biomes, Resources, OceanMountain",
```


Figure 55: Part of the state.json file.

The state file is stored in the JSON format. This provides strong flexibility and standardization, while it is slightly less efficient as less flexible options, the downsides are worth it since state is a relatively small part of the data.


```
"Biomes": {
  "$type": "System.Collections.Generic.Dictionary`2[[System.Int32, mscorlib],[terrainGenerator.Biome, Assembly-CSharp]], mscorlib",
  "0": {
    "Id": 0,
    "TileCount": 82712,
    "BiomeType": "Empty"
  },
  "1": {
    "$type": "terrainGenerator.OceanBiome, Assembly-CSharp",
    "Id": 1,
    "TileCount": 0,
    "BiomeType": "Ocean"
  },
  "2": {
    "$type": "terrainGenerator.SandBiome, Assembly-CSharp",
    "Id": 2,
    "TileCount": 0,
    "BiomeType": "Sand"
  },
```


Figure 56: state.json - The biome list.

One of the bigger things this file stores is a list of all biomes on the map. This is done to allow us to store additional information about each biome instance in a flexible way. This also allows us to bind tiles to biomes instead of biome types, allowing us to tell different biomes of the same type apart.

```
"Resources": {
  "$type": "System.Collections.Generic.Dictionary`2[[System.Int32, mscorlib],[terrainGenerator.Resource, Assembly-CSharp]], mscorlib",
  "0": {
    "Id": 0,
    "ResourceType": "Gold",
    "Tiles": [
      "436-185-0,2",
      "436-186-0,28",
      "437-184-0,23",
      "437-185-0,31",
      "437-186-0,38",
      "437-187-0,25",
      "438-185-0,43",
      "438-186-0,59",
      "438-187-0,35",
      "439-185-0,65",
      "439-186-0,72",
      "439-187-0,47",
      "440-186-0,53"
    ]
  },
  "1": {
    "Id": 1,
    "ResourceType": "Gold",
    "Tiles": [
      "445-192-0,32",
      "446-191-0,36",
```


Figure 57: state.json - The resource list.

Similar to how biomes are stored in the state, resources on the map are also stored here. They are stored with information about the resource itself, like type and id. Unlike biome, resource stores which tiles it can be found on in the state instead of storing it as information in the tile. The reason for this is that each tile can hold an arbitrary number of different resources, while a tile only belongs to one biome.

While it takes more space to store that a resource is on a given tile in JSON than in the grid, the grid is not flexible. This means that to allow one tile to list 4 different resources that can be found on it, all tiles need to reserve enough space for 4 resources. With most tiles not having any resources, that is a lot of wasted space.

```
1 reference
public string Version { get; set; } = CURRENT_VERSION;

18 references
public string Name { get; set; } = "ImABigDummyWhoForgotToNameMyMap";

15 references
public float OceonLevel { get; set; } = Tile.OCEAN_ALT_CUTOFF;
15 references
public float MountianLevel { get; set; } = Tile.MOUNTAIN_ALT_CUTOFF;
8 references
public int Width { set; get; } //width of grid
6 references
public int Height { set; get; } //height of grid
2 references
public int Seed { set; get; } //random seed


// Number of images the grid is stored in.
3 references
public int GridImgCount { set; get; }


// Controls what tile properties are saved and how many bits they each get.
[JsonProperty]
3 references
private Dictionary<TileProperty, int> TileStorageConfig { get; set; }
```


```
"Version": "v0.3",
"Name": "Test1",
"OceonLevel": 0.26,
"MountianLevel": 0.88,
"Width": 513,
"Height": 513,
"Seed": 9999,
"GridImgCount": 0,
"TileStorageConfig": {
  "Biome": 8,
  "Altitude": 9,
  "Temperature": 9,
  "Humidity": 9
},
```


Figure 58: Side-by-side of the MapState class and the corresponding JSON data.

Using JSON for this has many advantages, especially related to flexibility. C# provides ways to automatically convert code into JSON and back. If we add more properties to biomes, they will

automatically be saved on future maps without anything else having to be changed. Additionally, the program will still be able to read the old maps without issue, reducing the risk of making breaking changes.

Tile

Tile properties are stored in the map file. Because a map contains a lot of tiles, especially as the map size goes up, we need to try to keep the size each tile needs as low as reasonable without losing anything important. An especially big focus here was minimizing wasted space. On some maps, wind might play an important role, while on others, wind might not even be used.

```
70 references
public enum TileProperty {
    None,
    Biome,
    Altitude, Temperature, Humidity,
    WindDirection, WindMagnitude,
}
```

Figure 59: The TileProperty enum.

```
TileStorageConfig = new Dictionary<TileProperty, int> {
    { TileProperty.Biome, 6 },
    { TileProperty.Altitude, 8 },
    { TileProperty.Temperature, 6 },
    { TileProperty.Humidity, 6 },
    { TileProperty.WindMagnitude, 6 },
    { TileProperty.WindDirection, 6 },
},
```

Figure 60: An example of a map defining how many bits to use on what.

What we ended up with was a system where each map can set its own rules for what to save and how much space to give each attribute. This way, if wind is not important for a map, the map can be configured to not store it at all, while if wind is important and precise data is needed, a lot of space can be given to it. It can be as precise as needed without wasting space on any possible edge cases.

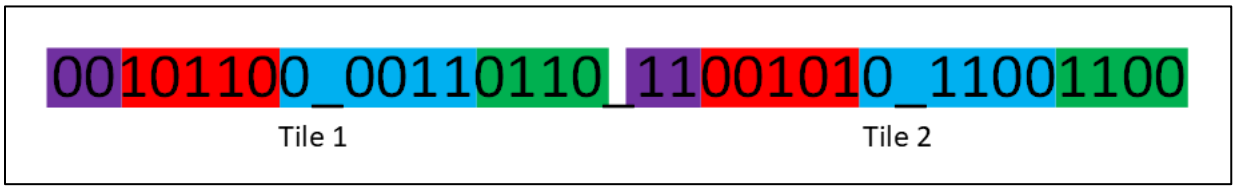


Figure 61: Diagram showing how different tile properties can be split over the bits of the file.

Each property is given a number of bits to store it in. The sum these property bits is all the space each tile gets. That way, we use as little space as we possibly can to store the data. The tiles are then stored in order. This format allows us to compress the data to an extreme level with minimal overhead. The downside to this format is that everything takes an exact amount of space, making it bad for things that do not always need the same amount of space, such as a list or name. For this reason we need a separate file for that information, in the form of the JSON.

The way tiles get encoded

When encoding a grid, we go through each tile in order, converting it to binary data and writing that to the file. Before encoding, we need to figure out how much space is needed and how to encode it. The process has been generalized to make it more readable and flexible. This way, adding more properties becomes easier. All that is needed is to add a new TileProperty and create functions to encode and decode them.

```
// Shift the number to make room for this property.
raw <<= bits;

// Encode the value and store it.
raw += (uint) (p.Encode(state, mask, t) & mask);

// Incease bitCount and check if it goes out of bounds.
rawWidth += bits;
```

Figure 62: This code moves the data to the left to make room for the next property, then adds it.

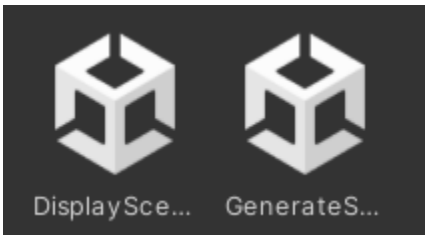
```
1 reference
private static int EncodeTemperature(IMapState state, int mask, Tile t)
    => (int) Math.Min(t.temperature.Scale(0, mask + 1), mask);
1 reference
private static void DecodeTemperature(IMapState state, int mask, Tile t, int val)
    => t.temperature = ((float) val).Lerp(0, mask);
```

Figure 63: Code that converts the temperature of a tile into a number, and applies the temperature read from file to a tile.

Each property has its own encoder and decoder that do the job using only the most important information. For the encoder, it is given the tile, the context, and how many bits it has been allocated, and returns the encoded value. For the decoder, it is given the same, but also the encoded value, and is expected to update the tile with the correct information.

When encoding the biome of a tile, each tile stores the id of the biome, with the information about each biome being stored in the state, thus avoiding repetition. When encoding values between 0 and 1, we multiply them by the highest number we can store, then round it. This way, the values fit easily into the space they are given. This has the downside of losing precision due to rounding, but the user can give more space to the value to get more precise numbers.

Unity scripts



Our unity code contains 2 scenes.

GenerateScene

This scene simply runs custom testing code. Its purpose is to call Program.Main() so that we can try out any needed code. The benefit of this is that it can run any code directly in unity. The drawbacks are that the code needs to be written and unity needs to rebuild the entire solution each time we make changes.

DisplayScene

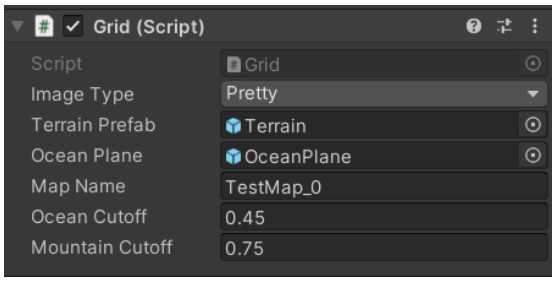
This scene is more complex. It allows us to display a map in 3D as well as edit and run the generators in real time.

Grid

Not to be confused with the Grid from the core library, the Grid script allows us to load and display maps in 3D.

```
/* ...
void LoadMap(string name) ...

/* ...
public void SetMap(GameMap map) ...
```

When first run, it will try to load the map with the name provided, if one is provided. Additionally, it can be called from other parts of the code and told to display a certain map, allowing us to use it in other parts of the program.



It has several fields to control how it displays a map. The image type field allows users to choose what map to show in 3D, for example the wind map. The ocean and mountain cutoffs control how much of the height to use for mountains and oceans in 3D space.

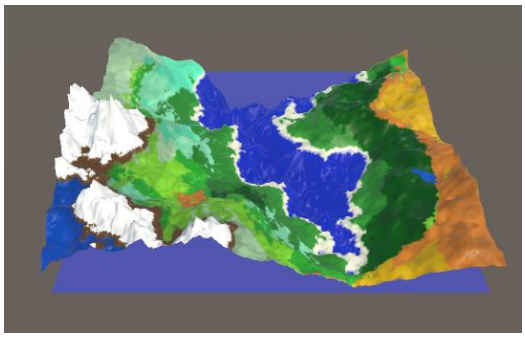


Figure 64: A map displayed in 3D

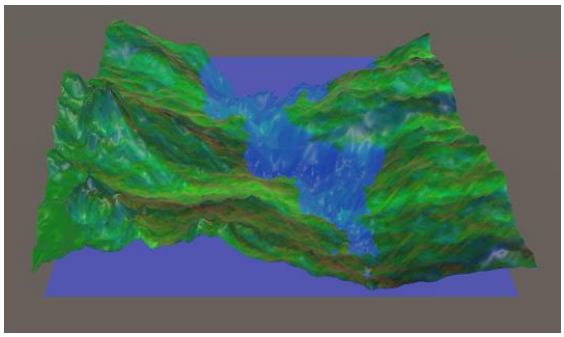


Figure 65: The same map displayed as a windmap instead.

RuntimeGenerator

The runtime generator allows us to edit our own map generator and run it in real time. This provides us with a flexible way to generate and try out generators and configurations.

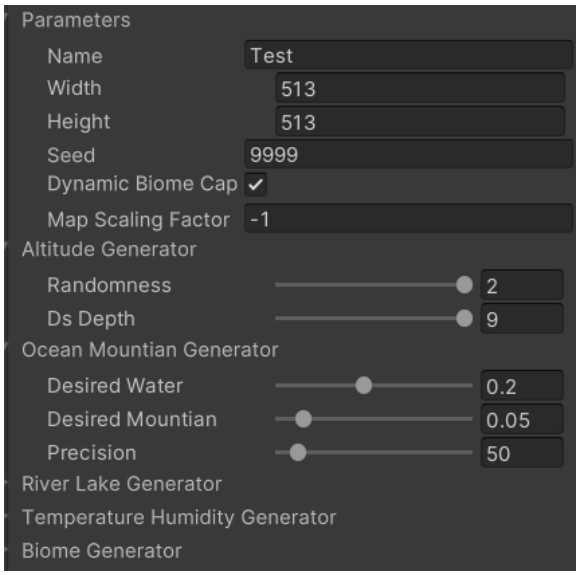


Figure 66: The first page for RuntimeGenerator.

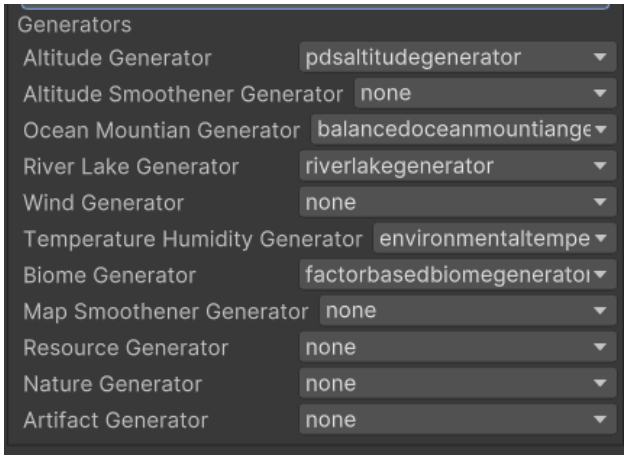
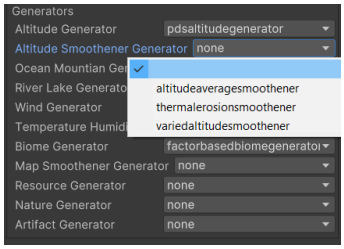


Figure 67: The second page for RuntimeGenerator.

It has 2 pages in the inspector.

The first page allows us to change the parameters of the map we are creating and alter the configurations of each step generator.



The second page allows us to swap out step generators or disable them all together. These lists of step generators are created on startup using reflection. That way, users do not need to manually add newly coded step generators.

Additionally, it adds a “generate” button to the UI that the user can press to generate a map using the generator they have designed.

Figure 68: A screenshot of the scene in action. Includes the RuntimeGenerator.

Source code file structure

Overall file structure:

- Assets
  - Code – Contains the unity related scripts.
  - Scenes – Contains the unity scenes.
  - Core – Contains the main code.
- Data
  - Maps – The maps that are created.
  - Prints – The images depicting how a map looks.
  - Test – Other various files, like benchmarking tests.

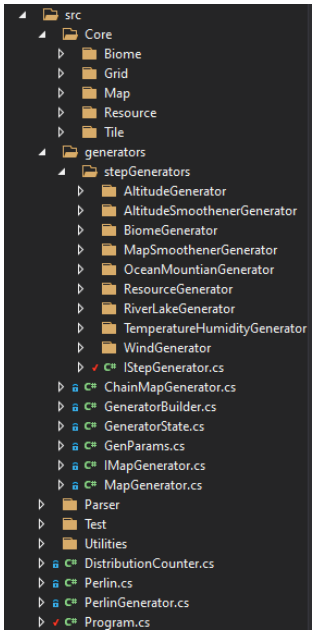


Figure 69: The source code folder for the core library.

The folder structure for the core library:

- Core – This contains the types that are fundamental for building our program.
  - Biome – Contains classes related to biomes, biome types, and information about these.
  - Grid – Contains our grid system, which we use to represent the tiles on a map.
  - Map – Contains classes for representing maps and information about the map.
  - Resource – Contains classes for resources, resource types, and the presence of a resource on a given tile.
  - Tile – Contains classes for representing tiles and the things contained inside them.
- Generators – Contains generators and generator related classes, such as parameters.
  - StepGenerators – Contains classes representing a step in the generation process.
- Parser – Contains classes used to convert our data to files and back to code.
- Test – Contains various classes and methods used to perform tests and measure performance of our program.
- Utilities – Contains various methods that perform common tasks, allowing us to make the rest of the code more readable and simple.

Progress

Structure

The overall structure of the program changed significantly throughout the project. When starting out, we made functions that would create the whole map in one go. Though this was only to experiment and get the hang of how things worked. The first proper design had each map generator be a class inheriting from a Generator base class. Each one would have all the parameters be sent in when creating them and they would generate the whole map in one go. Eventually we ended up with the component based model we have now, where we define different step generators that have their own configurations and create a map generator by combining different step generators.

If we had stayed with the old model, our program would look significantly different. Each generator would have to do every part of the process, and they would have to take all parameters for the whole process in one go. We would probably have a lot fewer generators due to all the changes we would need to do when adding a new one.

Approximating Perlin range

One issue with Perlin noise is that the exact range of the generated values are unknown. There is an upper and lower bound, but the values close to the bounds are so unlikely to occur that the generated values may be within a much smaller range [14]. This is an issue, as we would like to control what the highest and lowest points on the map are, to give the designers more control over the map they are making. Fixing this would also increase precision and save memory, as fewer bits would be allocated to altitude values that are never used.

If the entire noise map is generated all at once, we can simply search for the highest and lowest value and normalize the entire map based on these values. However, if we instead want to request one coordinate at a time, without knowing the values of the rest of the map, our only option is to guess or approximate the range. We experimented with multiple methods for this.

One method was to sample a fraction of the noise map and use it to get an idea of the distribution. Of course, the entire map could be sampled to find the actual lowest and highest value, but this would defeat the purpose, as the goal is to decrease computation time and only generate noise values when needed. The sampled points would have to be evenly spread out, to ensure that all the points are not in some local area of higher or lower elevation. Generally, these sampled points would be less extreme than the actual minimum and maximum values, as the more points you check, the more likely it is that one of them will hit a rare extreme. We did various tests to try and find the relationship between sampled values and the actual ranges of a noise map.

```
Max:      (-2.430, 2.430)
Actual:   (-1.857, 1.729)
Sample:   (-1.476, 1.386)
SAdjust:  (-1.697, 1.594)
CoefAdj:  (-1.727, 1.660) 0.263
CoefAdj2: (-1.639, 1.564) 0.171
CoefAdj3: (-1.965, 1.921) 0.513
```

Figure 70: An experiment on a 513x513 Perlin noise map, using a 16x16 sample.

Here "Max" is the mathematical lower and upper bounds. This is calculated based on the parameters, like the number of octaves and persistence. "Actual" is the actual lowest and highest generated noise values, while "Sample" is the lowest and highest points found by sampling. The last 4 lines are various attempts at interpolating between the sample values and the max theoretical values, to try to get closer to the actual values. The coefficient for interpolation is calculated using various methods. We struggled to find a rule that would work for varying dimensions, and overall found the random samples would vary greatly.

In the end we found this was out of the scope of this thesis. As we decided maps would have a pre-defined size, generating noise values one coordinate at a time became unnecessary. All our generators that use Perlin generate the entire noise map at once, but we still have working functions in the code to sample and generate noise values on the fly, should a user wish to use it that way.

Rivers and Lakes

Glaciers

We at one point had rivers capable of spawning from ocean and spreading upwards until hitting a mountain or local maxima. In the latter case, a glacier would generate as the river's source, similar to how a lake is generated, but with the directions flipped. This was eventually scrapped as we developed a more intricate biome and factor system. We felt glaciers were a more specific biome than lakes, and should be tied to factors like temperature, or be part of a separate step generator that manages water sources or sub-biomes. In the end we did not re-implement glaciers in any generators, but the biome still exists and can be easily added.

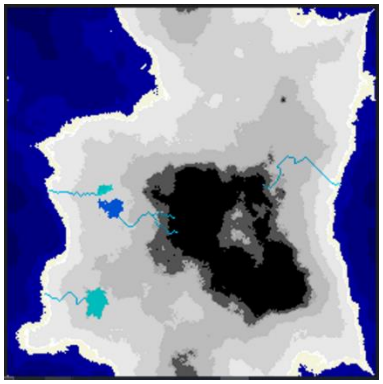


Figure 71: The lighter blue blobs are glaciers, acting as river sources.

Stretched Bezier curves

We also attempted to make the Bezier curves of rivers in RiverLakeGenerator wider by "stretching" some of the points away from the river. The idea was to find out how much we could customize the curves while keeping the general direction, shape, and natural look. This could allow us to make various river patterns, such as a river that has larger curves when it is wider, like in nature. However, we found that this was more challenging than expected, and was ultimately outside the scope of this project. The main issue was making it look natural and keeping the river flowing downwards at the same time. The ClosestOceanRiverGenerator allowed us to use some of this desired functionality, as the exact number of curves and curviness of each river could be specified and still look good. These

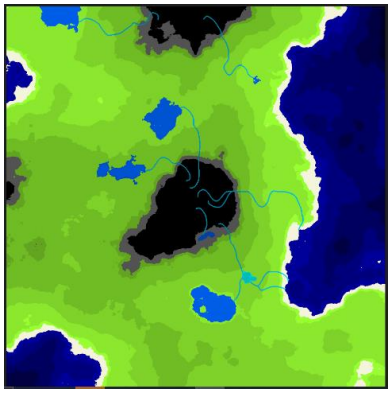


Figure 72: A map whose rivers have manually stretched bezier curves.

rivers had only a start point and an endpoint, rather than many points the river had to go through, so they were easy to customize. This stretching just did not meld well with rivers that search for a path downwards.

Saving and loading maps

The map saving system went through a lot of changes over time. Originally, the data was intended to be stored on a single image. This was achieved by looking at each pixel as an integer (4 bytes = 32 bits), while different properties of the tiles were stored on different parts of this integer. Initially, this worked out fine, but it quickly became limiting. The first thing we added was a separate JSON file for information about the map that was not tied to any tile. This includes things like map name, dimensions, and altitudes of ocean and mountain.

As more properties were added to tiles, the hardcoded encoding became impractical. We were limited to 32 bits total while we were testing many different attributes that needed to be stored, like wind, altitude, humidity, temperature, and biomes. To solve this, we abstracted the process and created the TileStorageConfig, a map setting that let us control how many bits to use for each property as well as which properties to store. This was configured for each map, allowing us to reserve more space for the properties we were testing.

We were still limited to a single image (32 bits per pixel) which would be bad for actual maps, since they might want precise data on all properties of the tiles. To solve this we changed the way data was stored so that if the total went over 32 bits, it would split the data into 2 images, if it went over 64 bits, it would split into 3 files, etc. This allowed us the space we would need for any case while only using more space if it needed. However, there was still a lot of potentially wasted space since it would have to use an extra image to fit the data, even if only one bit was needed. Additionally, having multiple images with the same name ending in a different number was needlessly messy.

After talking with the employer, we agreed to change their previous specification, which required the map to be stored in an image file. We started storing the data in a binary file instead, along with a separate image that shows what the map looks like, as well as the JSON. This way, the space used is only rounded to the nearest byte, and the entire grid is saved in a single file.

Unity integration

When first making the change from simply saving files to actually rendering in 3D using Unity, we started with a quick proof of concept. We would read the file and create a cube object for each tile. The altitude value was used to stretch the cube vertically, and the biome was used to determine its color. We also added a semi-transparent plane to represent the ocean level.

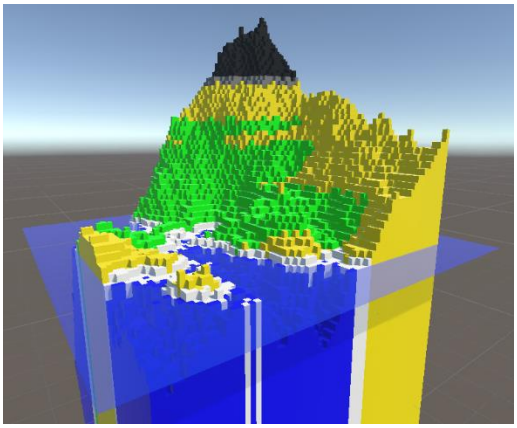


Figure 73: The first terrain we rendered in 3D.

We quickly realized that this was a massively inefficient method, as each tile requires Unity to create a separate game object, and the number of tiles increases exponentially with map size.

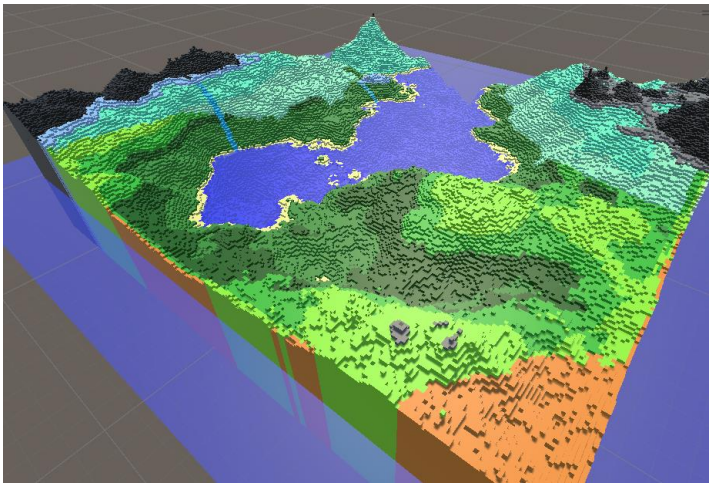


Figure 74: A larger map with one game object per tile. Though we liked the stylized look, this is not what the employer wanted, and it needed to be optimized.

We considered two options for rendering the terrain: creating a custom 3D mesh, and using Unity's terrain feature. We decided on the ladder, as it is specifically made for managing terrain and is easy to use. One downside was that the terrain heightmap had to be a square of size (2^n)+1, which we discuss later. We noticed the altitudes would appear blocky, but this became less apparent as map size increased and we added altitude smoothening steps.

Seeing the maps represented in 3D also made apparent certain issues we had not realized when looking only at 2D maps. One example of this was how rivers would sometimes flow upwards, or how the lakes would sometimes look unnatural. A lot of focus was then put into making rivers and lakes look and behave more realistically in 3D as well as 2D.

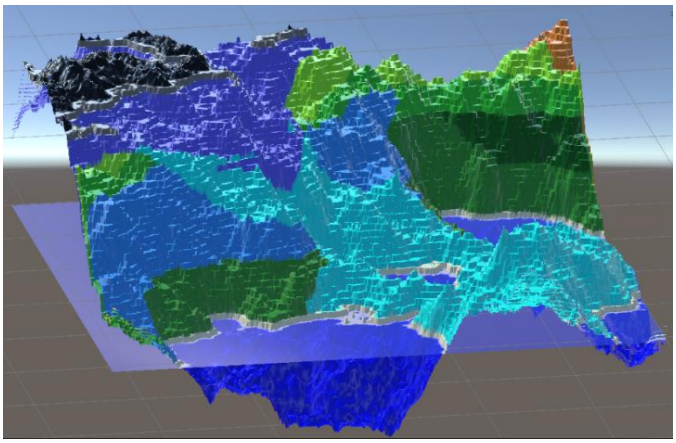


Figure 75: A very steep landscape made using Unity's terrain feature.

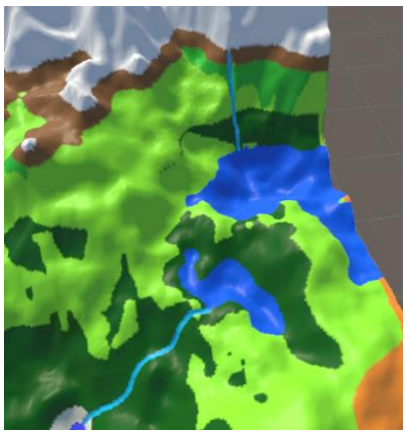


Figure 76: Two nearby lakes after many improvements to the river and lake generation. One is the result of a river, while another spawned by itself and overflows into a river.

Another issue we noticed was how mountaintops would often appear spiky and unnatural. This made us focus more on altitude smoothening, for instance with thermal erosion.

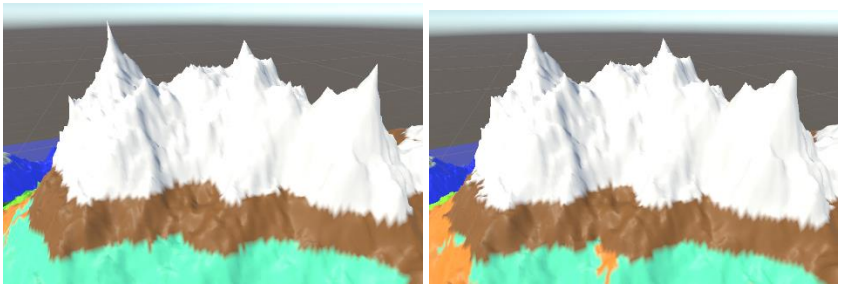


Figure 77: The same mountains with no smoothening vs with thermal erosion.

Non-square terrain in Unity

Terrain in Unity must be a square of size (2^n)+1 (ie. 65x65, 129x129, etc.). Initially, we had this as part of the map constraints. Part of our reasoning was that this was also a restriction of the Diamond Square algorithm, which is a prominent type of noise in many of our generators. We removed this restriction by making Diamond Square generate the smallest square that fit this prerequisite and would contain the given dimensions, then simply ignoring the values outside the dimensions. This allowed our grid generator to use any 2D dimensions, even non-squares. After removing this restriction for the maps, it was important that Unity could render them all. The solution was similar to our Diamond Square solution. We make a terrain with the smallest possible size that can contain the map, then set all tiles outside these dimensions to holes, preventing unity from rendering that area. We also make sure the position of the terrain and its texture accounts for this.

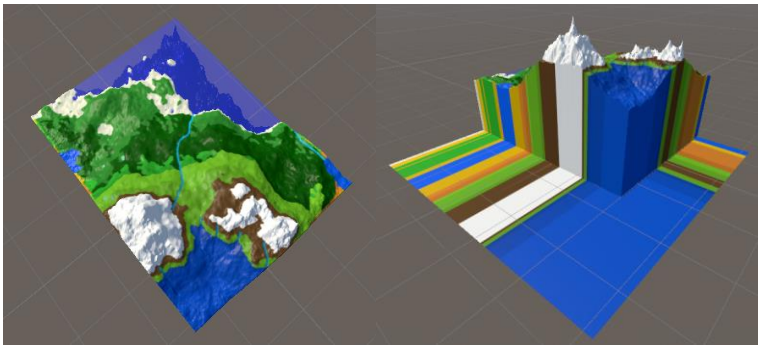
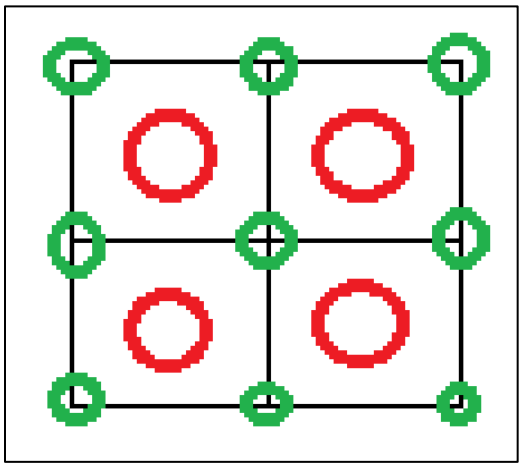


Figure 78: The same terrain with and without holes on out-of-bound tiles. The out-of-bound tiles all have altitude 0, the default value, and are outside the bounds of the biome texture.

Lining up texture with heightmap

One issue that persisted for a long time was the texture representing the biomes not lining up perfectly with the heightmap on the terrain in Unity. This was because textures for terrain should generally be of a size 1 lower than the heightmap in each dimension, such that each pixel can be placed between 4 heightmap points in the terrain.

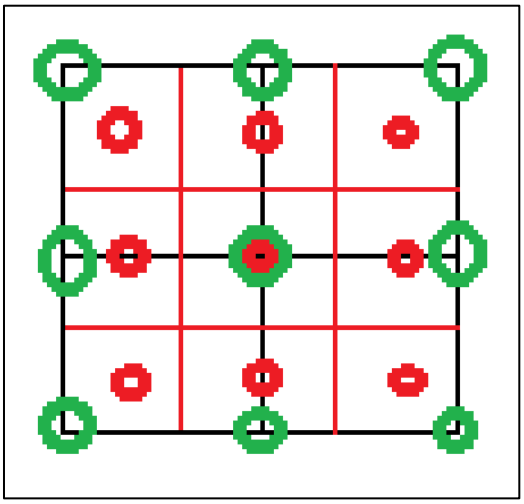


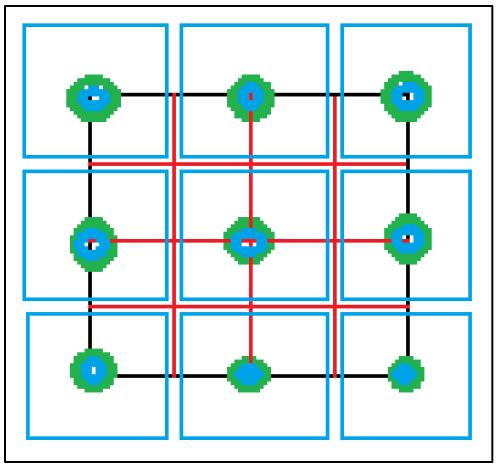
Here green circles represent a point in the heightmap, while red circles represent a pixel in the texture. The heightmap is 3x3, while the texture is 2x2.

However, since we want each point in the grid to have both an altitude and a biome, we have a texture whose size matches that of the heightmap, and is slightly squeezed to fit on the terrain. But this does not result in the center of each pixel being at the same point as the altitude point in the heightmap. This is because the altitude points along the edge are exactly on the edge, while the pixels along the edge have their center slightly withing the terrain. This results in a slight offset for every pixel except the one in the very center.

The red lines represent the boundaries between pixels, as they are no longer matching the lines between the heightmap points. The red circles are the centers of each pixel, and they are not matching the locations of the heightmap points as planned.

The solution to this problem was to edit the texture, as we can stretch or squeeze any size onto the terrain. The size of each side of the resulting texture is (n*2)-2, where n is the size of the side in the biome grid. It is essentially a texture with twice the size, each biome now being represented by 2x2 pixels, except the pixels along the edges are all removed. This results in the center of each of these 2x2 chunks, that each represent a tile's biome, matching the position of the corresponding heightmap z value.





The red lines represent the boundaries between actual pixels in the texture. The blue squares represent the biome data of 1 tile from the grid, each made up of 4 pixels. Along the edge, not all of these 4 pixels are contained in the terrain. The center of each of these "tiles", represented by blue circles, matches the position of the heightmap points.

This does result in the tiles along the edge of the map appearing smaller in 3D, but at large map sizes, this is almost imperceptible. An alternative solution would be to edit the heightmap by adding points around the edge, and matching their altitudes with the closest point. However, this would have resulted in the edge of the map suddenly appearing flat. We preferred to cut the edge of the map short, rather than distorting it.

7 Testing

Benchmarking

Sizes	129	257	513	1025	2049	4097	Notes
Altitude_Perlin	00:00:00.0404	00:00:00.1579	00:00:00.6308	00:00:02.5724	00:00:10.2935	00:00:41.2164	
Altitude_DS	00:00:00.0014	00:00:00.0059	00:00:00.0219	00:00:00.0988	00:00:00.4064	00:00:01.7963	
Altitude_PDS	00:00:00.0414	00:00:00.1617	00:00:00.6451	00:00:02.5819	00:00:10.3698	00:00:41.5147	
AltSmooth_ThermalErosion	00:00:00.0109	00:00:00.0349	00:00:00.1134	00:00:00.4236	00:00:01.7354	00:00:06.9923	
AltSmooth_Average	00:00:00.0239	00:00:00.1004	00:00:00.3724	00:00:01.5063	00:00:06.3431	00:00:25.5408	
AltSmooth_Varied	00:00:00.0374	00:00:00.1208	00:00:00.5046	00:00:02.0802	00:00:07.1933	00:00:30.7917	
OceanMountain_Raw	00:00:00.0044	00:00:00.0177	00:00:00.0594	00:00:00.1801	00:00:00.5907	00:00:02.2922	
OceanMountain_Balanced	00:00:00.0055	00:00:00.0167	00:00:00.0598	00:00:00.2344	00:00:00.9807	00:00:03.8557	
RiverLake_Closest	00:00:00.0034	00:00:00.0029	00:00:00.0109	00:00:00.0308	00:00:00.0283	00:00:00.3723	
RiverLake	00:00:00.0054	00:00:00.0187	00:00:00.0224	00:00:00.0428	00:00:00.0260	00:00:00.0204	
Wind_Flowing	00:00:00.0235	00:00:00.0841	00:00:00.3221	00:00:01.2856	00:00:05.1084	00:00:20.7990	
Wind_LocalTiles	00:00:00.0458	00:00:00.1857	00:00:00.7363	00:00:02.9863	00:00:11.7883	00:00:47.6435	
TemperatureHumidity_Perlin	00:00:00.0784	00:00:00.3138	00:00:01.2538	00:00:05.0217	00:00:20.0758	00:01:20.1058	
TemperatureHumidity_DS	00:00:00.0034	00:00:00.0119	00:00:00.0566	00:00:00.2498	00:00:01.0422	00:00:04.7275	
TemperatureHumidity_Environmental	00:00:00.6163	00:00:02.3647	00:00:06.1541	00:00:21.2583	00:01:27.7561	00:03:13.0464	Range: 160, Chunks: 2
TemperatureHumidity_Environmental	00:00:00.2316	00:00:00.8824	00:00:03.1257	00:00:11.3351	00:00:44.7974	00:01:59.7566	Range: 160, Chunks: 4
TemperatureHumidity_Environmental	00:00:00.2034	00:00:00.7604	00:00:02.8613	00:00:10.5237	00:00:41.6887	00:01:53.6192	Range: 160, Chunks: 6
Biome_Expanding	00:00:02.2652	00:00:08.4919	00:00:34.4579	00:02:17.3858	00:09:08.0226	00:37:22.5081	
Biome_FactorBased	00:00:00.0114	00:00:00.0319	00:00:00.1844	00:00:01.4159	00:00:17.9109	00:05:20.2790	
Biome_AltitudeBased	00:00:00.0074	00:00:00.0229	00:00:00.1583	00:00:01.0643	00:00:12.8843	00:03:42.5472	
Biome_OneDNoise	00:00:00.0498	00:00:00.2003	00:00:00.7396	00:00:03.3404	00:00:19.4218	00:03:29.6349	
Smoothening_CellularAutomata	00:00:00.2262	00:00:00.8338	00:00:03.3889	00:00:13.7728	00:00:54.1154	00:03:38.6501	
Resource_FloodFill	00:00:00.0029	00:00:00.0000	00:00:00.0000	00:00:00.0039	00:00:00.0040	00:00:00.0059	
Resource_Primitive	00:00:00.0014	00:00:00.0000	00:00:00.0000	00:00:00.0000	00:00:00.0009	00:00:00.0000	
Total	00:00:03.9422	00:00:14.8225	00:00:55.8806	00:03:39.3960	00:15:02.5848	01:05:47.7172	

Figure 79: A table listing benchmarking results.

To measure the performance of different step generators and ensure they were all reasonable, we ran benchmarking tests on them. The benchmarking consists of running every step generator one after another, and timing how long they take. The tests are done multiple times on multiple different map sizes. The times for each size are averaged to get a more accurate number with less variance.

By running each step on the same map, we reduce the chance of a test getting an unfair advantage. For example, a river generator may be faster on a map with more ocean. Some generators may still get some advantages, as what speeds up one step generator may slow down another, but this at least ensures they all got the same test. In addition to giving us an impression of how fast each generator is, we can also see how they scale with larger maps.

Some generators are relatively fast, while others take quite a bit of time. This is to be expected as some perform simple tasks, like generating Perlin noise, while others do much more complex jobs, like simulating spread and dispersion, which requires going over the map multiple times and doing costly operations. This data can help the user to pick between quick generators and more complex generators.

While performance was not a major focus of this project, we did some improvements where the performance was particularly bad and there were reasonable improvements we could do, such as chunking when dispersing humidity. Overall the performance is within what was deemed acceptable, but could be improved further for some step generators.

Determining map quality

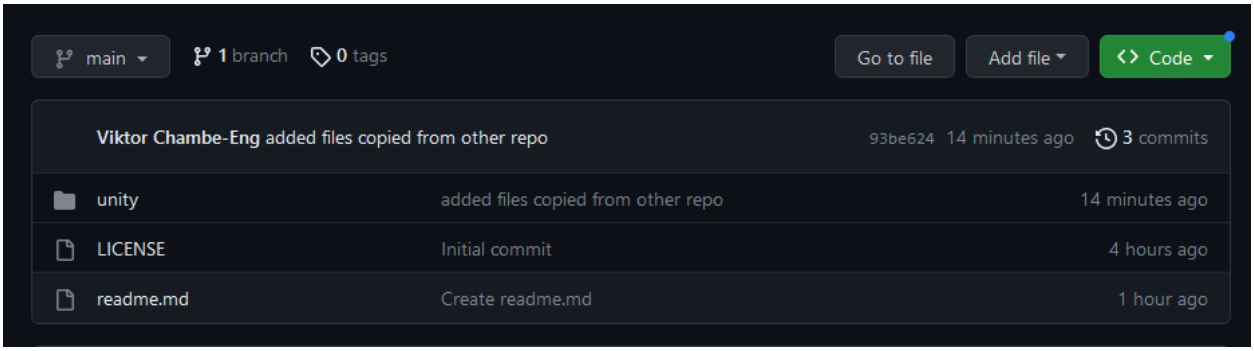
In addition to being fast, the output of the step generators also needs to be useful. The quality of a map is highly subjective and cannot be measured automatically the way performance can. As mentioned in section 2, the factors we deemed important were: balanced, strategic, informative, unpredictable, interesting, consistent, realistic, and stylized.

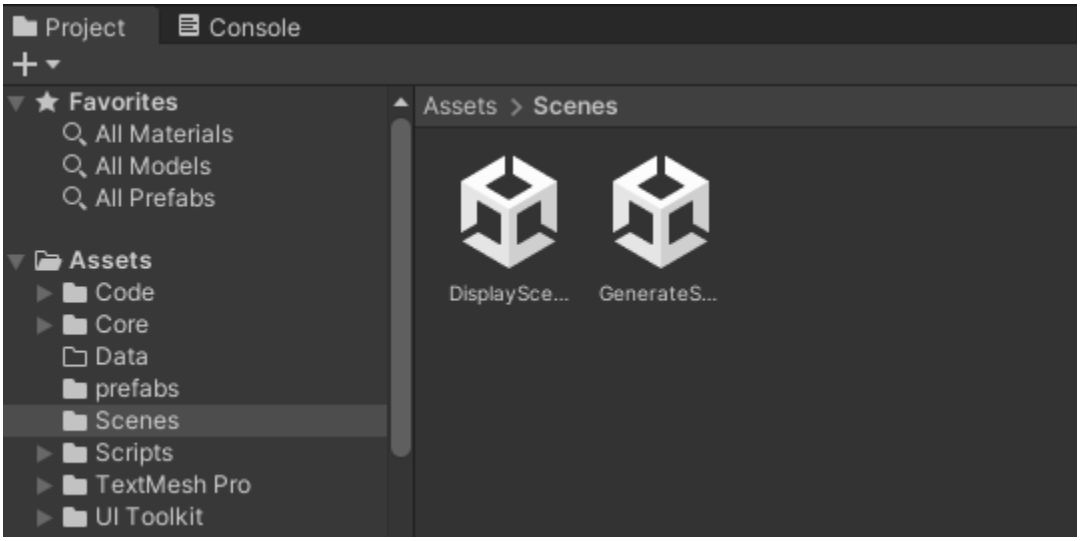
Throughout the developments process, we kept these factors in mind, and focused on creating a wide variety of step generators to make all of these factors a possible priority for the user. We discuss this further in the summary.

8 Deployment and integration

How to run the program:

1) Download the code from the public GitHub repository. (https://github.com/I-Love-Scratch/Procedural-Map-Generator)
2) Open the unity folder as a project in unity. (May take a while for unity to get everything ready the first time)
3) Go to the scenes folder and open DisplayScene.





9 Summary

Discussions

The goal of our project was to develop a terrain generation tool. It was supposed to fit the needs of the game by Progress Interactive throughout its development process, as well as be an accessible tool for anyone else who might need terrain generation for their game. We have met that goal. We have developed a functional and highly flexible procedural terrain generation tool. It is capable of generating a wide variety of maps, with little to no issues or artefacts. In the end, we made a highly flexible and expandable framework, with many example step generators ready for use. We believe this is a highly useful program, both for the employer and anyone else who might want to use it to generate terrain for their game.

Though the specifications of the project were difficult due to communication challenges with the product owner, we worked efficiently and consistently, while always focusing on maintainable and expandable code. We followed a strict schedule, and always worked on what was most crucial at the time. We often used peer programming, and when coding separately, followed strict repository conventions to ensure little to no conflicts between our code. We made a solid skeleton, then added as much meat to the solution as we could. There are some areas where our code could be improved and features could be added, but it is entirely functional. Overall, we are happy with the final product.

Evaluation

The product

Though the metrics we defined for what makes a good map are all highly subjective, we believe we have achieved them all to some extent with our implemented step generators. There is also a huge potential for users to expand on these and focus on the metrics that are most important to them.

We can generate terrain that has a realistic looking heightmap with PDSAltitudeGenerator and ThermalErosionSmoothener. We can simulate systems that mimic nature to place biomes with RiverLakeGenerator, FlowingWindGenerator, EnvironmentalTemperatureHumidityGenerator, FactorBasedBiomeGenerator, and BiomeBasedAltitudeSmoothener. For a more stylized look, users can smoothen the terrain with AltitudeAverageSmoothener, or make their own biome color palette.

Through our biome matrix system coupled with EnvironmentalTemperatureHumidityGenerator and FactorBasedBiomeGenerator, we can generate interesting maps that still follow consistent patterns. When you see a desert in a low area, you know you are in a place with high temperature and low humidity, and can make informed decisions about what is around you. You would know that there likely are no water sources nearby, and that you will not find cold biomes like tundra unless you travel far. Yet you do not know if you will find a savanna or a grassland, or even a mountain range with a great lake behind it, whose humidity could not spread past the peaks to reach the desert. This keeps the map informative yet unpredictable, and overall makes the terrain feel more alive.

How balanced and competitive a map is is difficult to determine without the actual game, but the fact that maps are easy to customize and iterate over should make it easy for a designer to balance. FloodFillResourceGenerator is a step generator that could greatly affect balance depending on what the resources do, and is simple to expand and tweak, by for instance adding a resource and changing how often and in what biomes it spawns.

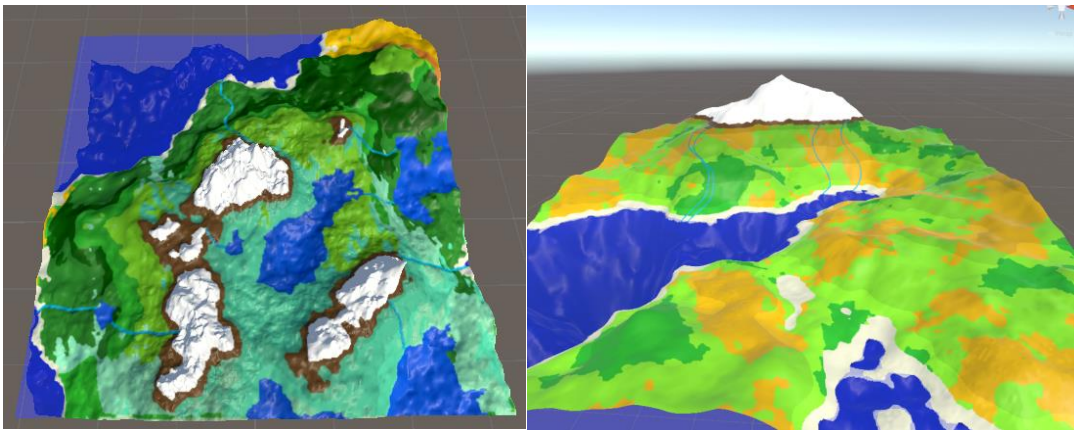


Figure 80: A more realistic map next to a highly stylized one.

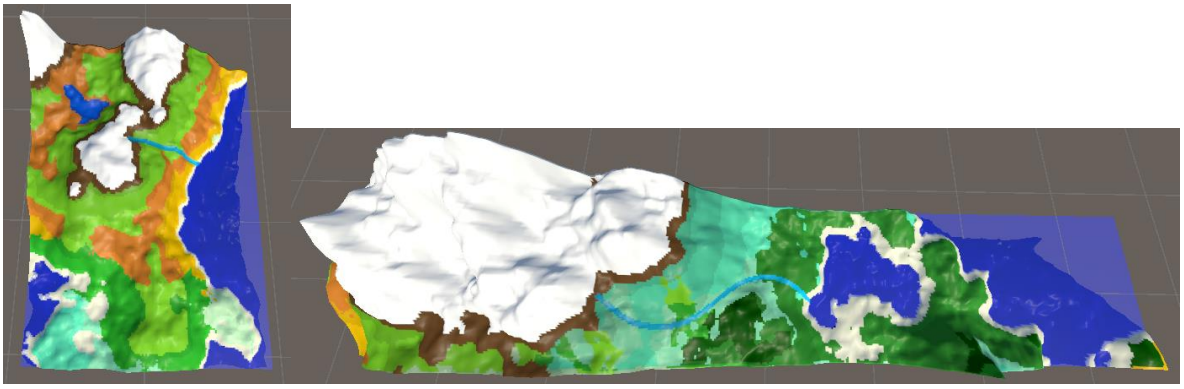


Figure 81: Two maps that demonstrate consistent rules yet have room for surprises.

The process

Although the scope and specifications of the project changed significantly throughout the development process, we feel we have been consistently productive, and always focused on functionality the client wants. Our time log and commit history can attest to that. Though meetings with the employer were infrequent, we always tried to get as much as possible out of our meetings with both the employer and advisor. We asked constructive questions and tried to nail down specifications and implementation details as quickly as possible, and always tried to get a date for the next meeting by the end. See our meeting logs for details.

For work distribution, while we specialized in different areas over time, we both kept a good grasp of the overall program and did equal amounts of work. We would casually discuss what we had done and what to do next every working day. Neither of us have any grievances with how or how much the other has worked.

Further work

Since the framework is flexible, there are many things we could expand on. We could add generators that do more game-specific things in the nature and artefact steps, if we had more details about the game. We could add more step generators for every step to make the tool more appealing and complete to other users. We could make support for alternative ways of storing the map data, like voxel grid or vector displacement field, more easily integrated. But there are some features we would prioritize over others if we had more time.

A major priority would be to fully integrate biome instances. Right now, most biomes only have one instance per map, even if there are clearly multiple blobs far away from each other. This is because when determining the biome of a tile through factors, we have no way of knowing what the surrounding tiles are. If we later try to group biomes up into instance, it raises questions about what constitutes an instance, as most biomes have some stray tiles close to the edge that to not connect to the rest. One solution could be to check the surrounding tiles within a certain range to group them up into one biome, but this would be computationally expensive.

For wind and sunlight simulation, our generators are hardcoded to only support certain directions. The ability to specify any angle would fit more with our flexible design philosophy, but would require major rewrites for those step generators.

There are sever tweaks we could make to the rendering in unity. For example, each lake could have its surface represented by a plane. The altitude would be easily found in the JSON file, but the exact boundaries of the plane would be more complicated, as they would have to perfectly match the shape of the lake. We could also give the user more lighting options, like showing the map at night or rendering shadows.

The runtime generator for unity could also use some improvements. Firstly, it is buggy. The inspector tool was not a big focus of this project, as it was mostly made to quickly try out different settings and step generators. When switching out step generators, you have to click out of the runtime generator then click back in for it to work. Secondly, there are a few functionalities it could use. It has no way to save the maps you generate or any way to use a map as a basis so you can improve an existing map. It also only works on MapGenerator, while using it to make ChainMapGenerator would be useful.

There are also generally some remnants of old code that are still cluttering the files. For instance, there are several fields in the BiomeTypeObject file that are only used in ExpandingBiomeGenerator, and should be separated. This does not affect functionality in any way but does make the code more confusing to read. If we had more time, we would focus on making the code more intuitive in certain places. We are however happy with the amount of comments and overall folder structure of the repo.

Finally, though most step generators are flexible and customizable, some lack options that we would have liked. Perlin noise is an example of this, as lacunarity and persistence are hardcoded. This was based on the testing we did ourselves, as different values often resulted in maps that were either too boring or too wild in their distribution. If we were to give the user the option to set these themselves, the generation may become much more risky, as certain parameters could result in unacceptable maps. Therefore, we would need to do testing to find the acceptable ranges we could allow the users to pick from.

Conclusion

Working on this bachelor project has been rewarding and challenging in many ways. We have gained valuable experience on how to manage long-term and large-scale projects. We have learned much about using technologies like C# and Unity3D on a professional level. Being able to contribute to both Progress Interactive's game and make a tool that could be genuinely useful to many creators has been incredibly rewarding. We are happy with our work and our results.

References

[1] J. Y. G. N. S. K. O. &. B. C. Togelius, "Search-based procedural content generation: A taxonomy and survey," 2011. [Online]. Available: https://nyuscholars.nyu.edu/en/publications/search-based-procedural-content-generation-a-taxonomy-and-survey. [Accessed 18 May 2023].

[2] chriskempke, "Heightmaps vs. Voxels," 2022. [Online]. Available: https://www.chriskempke.com/heightmaps_and_voxels/. [Accessed 21 May 2023].

[3] R. F. Bedrich Benes, "Layered data representation for visual simulation of terrain erosion," 2001. [Online]. Available: https://www.semanticscholar.org/paper/Layered-data-representation-for-visual-simulation-Benes-Forsbach/6b87f8be9e0392e084efe35b36038b9ad30b1897. [Accessed 20 May 2023].

[4] C. McAnlis, "HALO WARS: The Terrain of Next-Gen," 2009. [Online]. Available: https://www.gdcvault.com/play/1277/HALO-WARS-The-Terrain-of. [Accessed 21 May 2023].

[5] H. Kniberg, "Reinventing Minecraft world generation by Henrik Kniberg," 2022. [Online]. Available: https://www.youtube.com/watch?v=ob3VwY4JyzE&ab_channel=Jfokus. [Accessed 20 May 2023].

[6] S. Wolfram, "Statistical Mechanics of Cellular Automata," 1983. [Online]. Available: https://web.archive.org/web/20130921060232/http://www.stephenwolfram.com/publications/articles/ca/83-statistical/. [Accessed 21 May 2023].

[7] W. B. Dev, "Lazy Flood Fill | Procedural Generation | Game Development Tutorial," 2021. [Online]. Available: https://www.youtube.com/watch?v=YS0MTrjxGbM&ab_channel=WhiteBoxDev. [Accessed 18 May 2023].

[8] K. Perlin, "An image synthesizer," 1985. [Online]. Available: https://dl.acm.org/doi/10.1145/325165.325247. [Accessed 20 May 2023].

[9] S. Sell, "GPU Accelerated Diamond-Square Generation," 2023. [Online]. Available: https://www.vertexfragment.com/ramblings/diamond-square/. [Accessed 18 May 2023].

[10] Microsoft, ".NET Platform," 2023. [Online]. Available: https://github.com/dotnet?WT.mc_id=dotnet-35129-website. [Accessed 21 May 2023].

[11] Jead, "What subscription tiers are available?," 2023. [Online]. Available: https://support.unity.com/hc/en-us/articles/208610336-What-subscription-tiers-are-available-. [Accessed 21 May 2023].

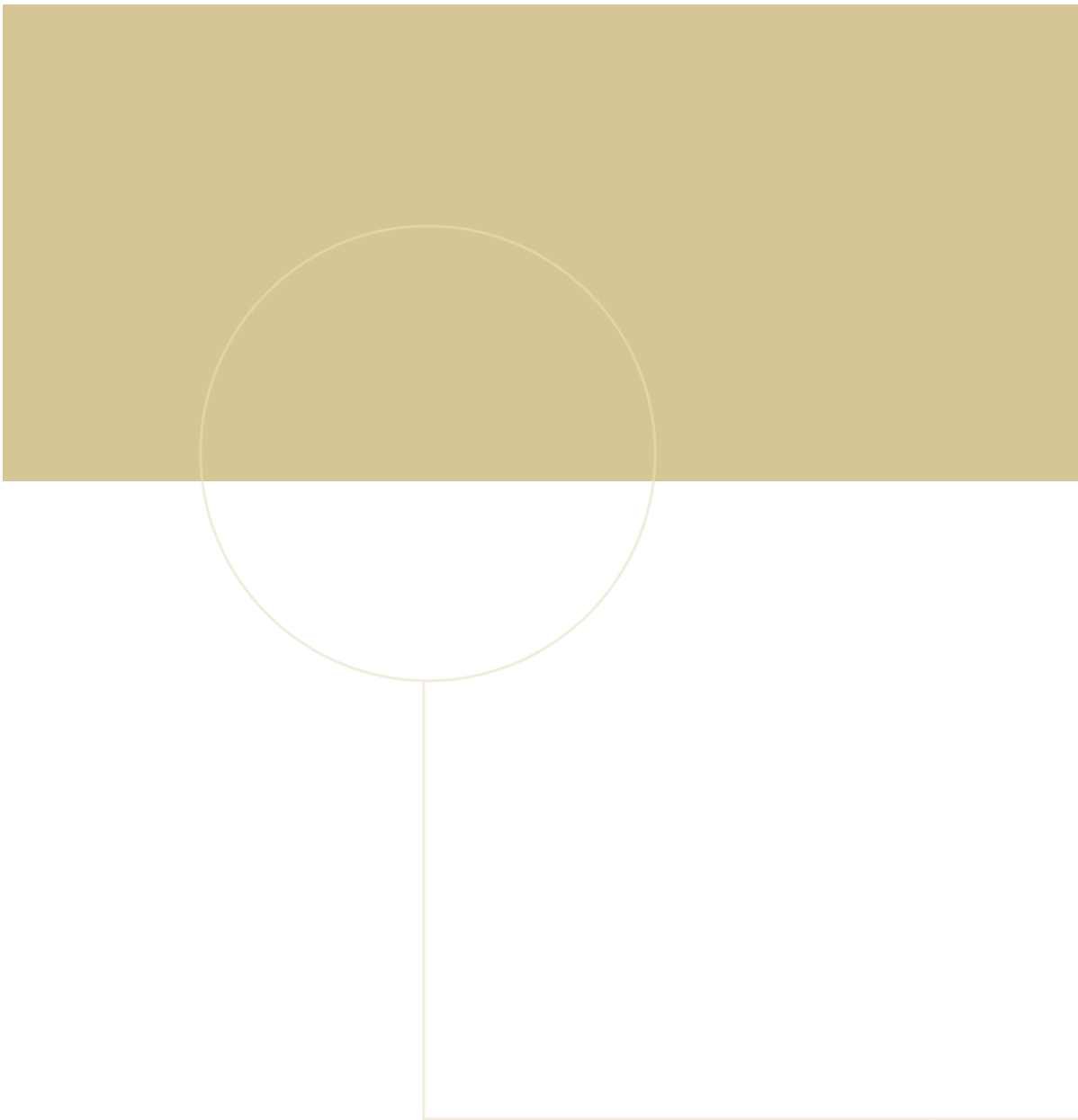
[12] Discord, "Permissions," 2023. [Online]. Available: https://discord.com/developers/docs/topics/permissions. [Accessed 21 May 2023].

[13] W3School, "JSON - Introduction," [Online]. Available: https://www.w3schools.com/js/js_json_intro.asp. [Accessed 21 May 2023].

[14] R. Chen, "The range of Perlin noise," 2017. [Online]. Available: https://digitalfreepen.com/2017/06/20/range-perlin-noise.html. [Accessed 21 May 2023].

Attachements

how_to_access_repo.txt	instructions on how to access the github repo
commit_history.txt	a text printout of every commit we made
MeetingLogs.pdf	logs taken during all our meetings
OverviewOfProgress.pdf	an overview of what we worked on each week
ProjectAgreement.docx.pdf	the signed project agreement with the employer
ProjectPlan.pdf	the project plan we made in January
TimeLog.pdf	an exstensive log of how many hours we worked each day



NTNU

Kunnskap for en bedre verden