

Universidade do Porto

Faculdade de Ciências

Mestrado em Ciência de Computadores

Procedural Generation of Terrains with User Defined Premises

André Ministro Tavares

Supervisor:

Verónica Orvalho

*A thesis submitted in fulfillment of the requirements
for the degree of Master of Science
at*

Departamento de Ciência de Computadores



Porto, September of 2021

Acknowledgements

This thesis is the result of my passion for Computer Graphics and the generation of procedural content, however it would not have been possible without the support of everyone around me.

I want to thank my family and friends for keeping me sane during the time I spent working in this thesis, who accompanied and helped me through all the times.

I also want to thank Verónica Orvalho who encouraged me to choose the topic I enjoyed and for supervising and guiding my work, as well as all the teachers I encountered throughout my education and may or may not have influenced me to follow the path that I chose. Thank you.

Abstract

Creating a full virtual landscape is a non-trivial task. Having a wide number of artists carving and populating a terrain is usually a very expensive way of accomplishing it, and this problem is tackled with the use of Procedural Generation. However, this is also a complicated subject that is still under research, and more methods and approaches to this task are always being developed, to improve how realistic the results are, improve the performance of the generation tools and even just invent new ways of controlling the generated terrain to fit the necessary needs.

Procedural Generation is an area of Computer Graphics that is increasingly being explored due to the potential of creating content for various forms of media, such as the entertainment industry, urban planning, education, among others. The gain in popularity of this technique is due to the ease and speed of creating a large amount of content of the same type automatically, always guaranteeing different results in all iterations. Content generally produced by Procedural Generation include natural or artificial environments, 3D models of various types, such as vegetation or objects to populate a certain area, or even the creation of complex structures such as buildings or vehicles.

In this line of thought, this thesis will have a greater focus on presenting a terrain procedural generation framework that generates surfaces from a 2D user sketch and can serve as a base for the implementation of any media application. The framework will be created having in mind it needs to produce distinct results from user to user. It also needs to be easily expandable and interactive so it makes the task of using the framework as a base to create terrains or even use the terrain generation method in other projects, easy and straightforward.

We studied pioneering and State of the Art approaches and methods to procedurally generate terrains to understand what are the best choices and solutions for our framework. We then present the solutions found and how they work, and give a detailed explanation on the implementation of them in the *Unity Engine*. We later assess our results and validate them based on the goals we set. Finally, we discuss what other future additions could make the framework better and more robust.

Keywords: Framework, Terrains, Procedural, Generation, Sketch, User.

Resumo

A criação de uma paisagem virtual completa não é uma tarefa trivial. Ter um elevado número de artistas a esculpir e povoar um terreno costuma ser uma maneira muito dispendiosa, e esse problema é resolvido com o uso da Geração Procedimental. Contudo, esta também é uma tarefa complicada, ainda em desenvolvimento, e cada vez mais métodos e abordagens estão a ser desenvolvidos, para melhorar o realismo dos resultados, melhorar o desempenho das ferramentas de geração e até mesmo inventar novas formas de controlar o terreno gerado para atender às necessidades necessárias.

Geração Procedimental é uma área da Computação Gráfica cada vez mais explorada devido à potencialidade de criação de conteúdo para variadas formas de media, como a indústria do entretenimento, planeamento urbano, educação, entre outros. O ganho de popularidade desta técnica deve-se à facilidade e rapidez de criar uma grande quantidade de conteúdo do mesmo tipo de forma automática, sempre garantindo resultados diferentes em todas as iterações. Conteúdos geralmente produzidos por Geração Procedimental incluem ambientes naturais ou artificiais, modelos 3D de vários tipos, como de vegetação ou objectos para povoar determinada área, ou até mesmo criação de estruturas complexas como edifícios ou veículos.

Nesta linha de pensamento, esta tese terá um foco maior em apresentar uma framework de geração procedimental de terrenos que gera superfícies a partir de um esboço 2D feito por um utilizador e pode servir de base para a implementação de qualquer aplicação de media. A framework será criada tendo em consideração que terá que produzir resultados distintos de utilizador para utilizador. Também precisa de ser facilmente expansível e interativo para que a tarefa de usar a framework como base para criar terrenos, ou mesmo usar o método de geração de terrenos em outros projetos seja fácil e direto.

Estudámos abordagens e métodos tanto pioneiros como o estado da arte de geração de terrenos procedimental, para entender quais são as melhores escolhas e soluções para a nossa *framework*. De seguida, apresentamos as soluções encontradas e como funcionam, e damos uma explicação detalhada sobre a implementação delas no *Unity Engine*. Posteriormente, avaliamos os nossos resultados e procedemos à sua validação com base nos objectivos estabelecidos. Finalmente, discutimos que outras futuras adições poderam tornar a *framework* melhor e mais robusta.

Palavras-Chave: *Framework*, terrenos, procedimental, geração, esboço, utilizador.

Contents

Abstract	iii
Resumo	v
Contents	x
List of Tables	xi
List of Figures	xvi
Acronyms	xvii
1 Introduction	1
1.1 Motivation	1
1.2 Problem and Goal	2
1.3 Objectives	3
1.4 Main Contribution	3
1.5 Applications	4
1.6 Outline	5
2 Related Work	7
2.1 Surfaces	7
2.1.1 Stochastic Approach	8
2.1.2 Simulation Approach	11

2.1.3	Learning Approach	14
2.1.4	Sketch Approach	15
2.2	Vegetation	16
2.2.1	L-Systems	16
2.2.2	Generation Methods	17
2.3	Networks	18
2.3.1	Roads	18
2.3.2	Rivers	19
2.4	Conclusion	20
3	Framework Overview	23
3.1	Architecture Overview	23
3.2	Framework Features	24
3.2.1	User Sketch	24
3.2.2	Voxel	26
3.2.3	Chunk	27
3.2.4	Terrain	28
3.3	Noise	39
3.3.1	Perlin Noise	40
3.3.2	Simplex Noise	45
3.4	Clouds	49
3.5	Terrain Visualization	50
3.6	Conclusion	51
4	Framework Implementation	53
4.1	Unity Overview	55
4.2	User Sketching Phase	56
4.3	Interpreting the User Sketch	57
4.4	Determine the Terrain Layout	59

4.5	Generate Terrain	62
4.5.1	Simplex Noise	64
4.5.2	Column Generation	68
4.5.3	Mesh Generation	70
4.5.4	Vegetation and Cave Systems	72
4.6	Cloud Generation	74
4.7	Terrain Visualisation	74
4.8	Conclusion	76
5	Results and Validation	81
5.1	Terrain Results	81
5.1.1	Water Masses	81
5.1.2	Hot and Arid Desert	82
5.1.3	Cold Desert	83
5.1.4	Temperate Forest	84
5.1.5	Taiga	85
5.1.6	Tropical Forest	86
5.1.7	Alpine Tundra	87
5.1.8	Savannah	88
5.1.9	Combination of all biomes	89
5.2	Terrain Evaluation and Validation	91
5.2.1	Validation on Terrain Size Coherence	91
5.2.2	Validation on Terrain Uniqueness	94
5.2.3	Validation on Framework Expandability and Versatility	95
5.2.4	Validation of the Accuracy on the Realism of the Terrain	96
5.3	Conclusion	97
6	Conclusion and Future Work	99
6.1	Conclusion	99

6.2	Future Improvements	100
6.2.1	Implementation of an Ecosystem	100
6.2.2	Implementation of the Marching Cubes Algorithm	101
6.2.3	Implementation of Weather and a Night and Day System	102
6.2.4	Implementation of the City "Biome"	102
6.2.5	Optimization of Execution Times	103
6.3	Final Thoughts	104
Bibliography		105

List of Tables

4.1	Biome Stone Values	63
4.2	Tiles used for each block present in this framework with its respective solidity . .	77
4.3	Water Masses Base Values	78
4.4	Hot Desert Base Values	79
4.5	Cold Desert Base Values	79
4.6	Taiga Base Values	79
4.7	Alpine Tundra Base Values	79
4.8	Temperate Forest Base Values	79
4.9	Tropical Forest Base Values	79
4.10	Savannah Base Values	79
5.1	Execution times, in seconds, for the generation of the water biome in each terrain size.	91

List of Figures

2.1	Fractal Mountain with River	8
2.2	Voronoi Diagrams	9
2.3	Voxel Structures 1	10
2.4	Voxel Structures 2	11
2.5	Terrain Layers	12
2.6	Water Erosion	12
2.7	Biomes representation	13
2.8	Terrain Synthesis from a Sketch and DEM	15
2.9	3D Sketches	16
2.10	Growing Corn Model	18
2.11	City Creation Phases	19
2.12	Generated River with Audiber and Belhadj Method	20
3.1	Framework's Architecture Overview	24
3.2	Sketch Example	26
3.3	Voxel Preview	26
3.4	Chunk Preview	28
3.5	Water Mass Example	31
3.6	Hot and Arid Desert Example	32
3.7	Cold Desert Example	33
3.8	Temperate Forest Example	34

3.9 Taiga Example	35
3.10 Tropical Forest Example	36
3.11 Alpine Tundra Example	37
3.12 Savannah Example	38
3.13 Tile Map	39
3.14 2D Perlin Noise	40
3.15 Graph with a Discrete Noise Function	40
3.16 Graph with a Continuous Noise Function	41
3.17 Perlin Noise 1D Function as the result of the sum of 6 continuous noise functions	41
3.18 Regular Static	42
3.19 Effect of Persistence on Perlin Noise output	43
3.20 Linear Interpolation Example	44
3.21 Cosine Interpolation Example	44
3.22 Perlin Noise 2D Function as the result of the sum of 6 continuous noise functions	45
3.23 Tetrahedron	46
3.24 Comparison between normal Perlin Noise and Perlin Noise that used a fall-off function instead of interpolation	47
3.25 2D Simplex Noise	47
3.26 Seed Choosing	49
3.27 Clouds Image	50
4.1 Framework sequence of events	54
4.2 Sketch UI	56
4.3 Inspector View on the TerrainManager GameObject	58
4.4 Inspector View on Image Reader GameObject	59
4.5 Part 2. Sketch Interpreter	59
4.6 Scheme of the Chunks layout	60
4.7 1 Chunk Highlighted	60

4.8	Inspector View of Chunk Prefab	61
4.9	Part 3. Determine Terrain Layout	62
4.10	Fall-off functions	65
4.11	1D Simplex Noise	66
4.12	Maximum possible values for the fall-off function in a triangle	67
4.13	Cube split into 6 tetrahedra	68
4.14	Mesh Rendering Example	70
4.15	Example of Unwanted meshes being rendered	71
4.16	Part 4. Generate the Terrain	73
4.17	Clouds GameObject Inspector	74
4.18	Player GameObject Inspector	75
5.1	Water Terrain 128	82
5.2	Hot Desert Terrain 128	82
5.3	Hot Desert Terrain First Person View	83
5.4	Cold Desert Terrain 128	83
5.5	Cold Desert Terrain First Person View	84
5.6	Forest Terrain 128	84
5.7	Temperate Forest Terrain First Person View	85
5.8	Taiga Terrain 128	85
5.9	Taiga Terrain First Person View	86
5.10	Tropical Forest Terrain 128	86
5.11	Tropical Forest Terrain First Person View	87
5.12	Alpine Tundra Terrain 128	87
5.13	Alpine Tundra Terrain First Person View	88
5.14	Savannah Terrain 128	88
5.15	Savannah Terrain First Person View	89
5.16	Example Sketch Orthogonal View	89

5.17 Example Sketch Top	90
5.18 Example Sketch Inside	90
5.19 Example Terrain Wall 32	92
5.20 Example Sketch 512	93
5.21 Example Terrain 512	93
5.22 Different Seeds Comparison 1	94
5.23 Different Seeds Comparison 2	95
6.1 Valheim Example	101
6.2 L-System buildings	103

Acronyms

1D	1-Dimension	FCUP	Faculdade de Ciências da Universidade do Porto
2D	2-Dimensions	GAN	Generative Adversarial Networks
3D	3-Dimensions	PG	Procedural Generation
CG	Computer Graphics	RGB	Red Green Blue
DCC	Departamento de Ciência de Computadores	UI	User Interface
DEM	Digital Elevation Model		

Chapter 1

Introduction

Procedural Generation is a key approach to create content efficiently for media applications. Generation of landscapes is usually a common task, since the work involved for an artist to create a terrain mesh, depending on the size, is absurdly high. Since the point of generating content procedurally is to reduce the work, and even the budget, for a certain project, it is rapidly gaining traction in the industry. One issue related to it, is the difficulty for a user to generate terrain procedurally, defining the land as he wants, without some knowledge on the area. We studied different techniques from the area of terrain procedural generation and present a framework that based on user input, generates a terrain accordingly. This chapter will provide an introduction to the thesis, giving the motivation behind it, stating the problem at hands and the main goals for the thesis' project, as well as the main contribution it provides. We will also provide an overview on the structure at the end of the chapter.

1.1 Motivation

Creating a landscape in the field of Computer Graphics, depending on the goal of the terrain, is not an easy task. It can be done by artists specialized in level design, by sculpting the mesh and populating it accordingly with the necessary assets, adequate to the objective of that surface. This, obviously, is a task completely subjective in terms of time and work involved, depending on the size of the terrain and how many artists are available to work on it. This is where Procedural Generation is useful. By generating a mesh using a logic system, adapting it to its goals, one can create a very detailed surface, that can even be made to look as realistic as possible, and can be used as a basis for artists to improve on. When used in video games or even in the film industry, it has amazing results, either when creating out-of-this-world scenarios or even authentic lifelike terrains. This is because Procedural Generation provides a sense of "randomness" and novelty very much appreciated in the industry. However, generation of terrains is never truly random because the approaches it uses have a set of premises decided by the developer which are the foundation of all the content the method creates. The premises act as construction rules

and constraints, which are manipulated through input parameters. Of course all this sense of novelty and randomness can, in reality, provide the opposite effect to the user after some time experiencing it, because as was said previously, nothing is actually random and in the long run, users start noticing the patterns and "rules" the world uses. It is in the hands of the developer to create a diverse experience. The first content to be procedurally generated were plants, by Lindenmayer [21]. Of course, this sparked the interest in other authors and the demand for generating surfaces in a procedural way, started early with the contributions of Mandelbrot in [24]. Many other approaches to do so, were developed in the following years. Stochastic approaches, with early work from authors like Prusinkiewicz and Hammel [36], from Olsen [30], using new approaches to divide the terrain, or Marak et al. [26], using a grammar-based method. Simulation approaches gained a lot of traction because they were the best at recreating lifelike scenarios, because of the use of algorithms that tried to recreate real life processes, like erosion, demonstrated by Roudier et al. in [40] or Št'ava et al. in [44]. Other authors, like Cordonnier et al., in [6] try to simulate entire ecosystems, and by doing so, complex algorithms carve the terrain accordingly. Recent approaches use real life terrain data taken from satellite imagery and Generative Adversarial Networks to generate realistic surfaces, as was done by Panagiotou and Charou in [31]. Finally, another valid contribution that gives control to the user generating the terrain, is by doing a sketch and use it as reference to create the landscape. With contributions from Yin and Zheng, in [47], using 2D sketches, or even in 3D, as was demonstrated by Becher et al. in [2].

This "sketch" notion is what incited us to develop this project, as it is a way to give a user the possibility of editing a terrain layout before it is generated, in a different way to what was already accomplished in previous works. It can contemplate the user choices and thus create full landscapes completely unique from user to user, where the sketch dictates how the terrain as a whole looks like, as opposed to translate only into a feature or characteristic. The next section will detail the problem at hands and how we want to solve it.

1.2 Problem and Goal

The main problem this thesis is aiming to cover and present a solution to is the lack of ability for a user to create a procedurally generated landscape with the characteristics and topography desired, without having the knowledge about the methods and algorithms regarding procedural content generation. Usually, procedurally generated terrains are used and created in video games or other forms of software, and users need to adapt to the construction rules and constraints established by the developers.

The final goal is developing a framework where a user sketches in a 2D canvas the desired terrain layout, using colours. Colours will represent certain regions, all with different characteristics, and the system will create the terrain as a 3D mesh with a coherent height map and biodiversity associated with the respective region the user chose for each determined position.

Of course, being a framework, the final goal also implies that it can be used as the first building step on a new application that needs a terrain generator, or can be used to generate a single terrain, ordered as the developer wants, to be the grounds of some media output.

1.3 Objectives

During the development of the thesis, a set of objectives were defined to guide the following chapters:

- Study of the previous work done in procedural generation of terrains and related structures;
- Create a solution based on the related work, trying to understand what are the best approaches to what is trying to be achieved;
- Implement the solutions and algorithms on a framework, in a straight-forward way, so it can easily be extended;
- Analyse the final results the framework outputs and validating them based on the previously set goals;
- Explore interesting solutions that could be implemented to increase the value of the framework.

1.4 Main Contribution

The main contribution of this thesis is the development of a terrain procedural generation framework that:

- lets users generate **unique**, **coherent** and **diverse** landscapes, with various biomes, with a degree of **realism**, based on a 2D **sketch**;
- is easily **expandable** and can be used as a basis for other projects;
- is **user-friendly** and **interactive**, as it provides a **UI** to produce the sketch along with the terrain generation method

As a result, this work speeds up the process of generating unique terrains with layout and topography decided by the user, and even allows to implement a video game or digital interaction on top of this work, as many structures and decisions were made with this goal in mind, diminishing the work and time consuming process of creating a terrain generator from scratch.

1.5 Applications

As was already mentioned, the main applications of terrain procedural generation is in the entertainment industry, in films and video games. Both industries, especially video games, are growing tremendously in the last years and we can see a lot of successful titles implement landscapes that are generated procedurally. *Minecraft* (2011), *Banished* (2014), *No Man's Sky* (2016) or *Valheim* (2021) are good examples of games which implement procedural terrains, because they can produce different and completely unique worlds every time a new one is generated, and although they are essentially the same in terms of what they offer, they have completely different topography and territorial ordering. In *Minecraft's* case, it even is an infinite terrain, so a user can keep walking forever and never once be in two locations that are visually equal. In the movie industry, not only landscapes are sometimes procedurally generated, as it is in the *Mandalorian* (2019) series, but also other forms of content. For example, in Peter Jackson's *The Lord of the Rings* films, the fighting armies composed of thousands of soldiers were generated using **PG**.

However, we need to make an important distinction. The games mentioned, have terrains that are generated in **real-time**, in other words, they are computed from scratch and rendered at run-time. Obviously, generating something in **real-time** is different from using a previously procedurally generated content. Video game studios use their proprietary procedural generation tools to generate terrains ahead of time and save them, to later be imported to a scene and be tweaked and edited by artists. Of course the goal of each type of generation is completely different, as **real-time** generation can be affected by user input and can produce more interactive results, as opposed to **offline** generation, that can be perfected and worked on until it is ready to be used.

Although the major focus of discussion is about the mentioned industries, Procedural Generation also has uses outside of entertainment. Virtual simulations can be used by the military, either for flight or infantry training, by generating terrains based or not in real world data, so logistics and planning can be made comfortably while giving the most realistic experience for an accurate military guidance. Some companies like Bohemia Interactive Simulations¹ specialize in this field. Other companies specialize in landscape rendering, but more focused on the demonstration, urban planning and even education, like *Terragen*², from PlanetSide Software. This solution is even used in TV productions, game development, VR environments, museum exhibits or documentaries.

¹<https://www.bisimulations.com/>

²<https://planetside.co.uk/>

1.6 Outline

The thesis will have the following structure:

Chapter 2: An overview of the State of The Art algorithms and methods to procedurally generate terrains, vegetation and networks in order to fully understand procedural generation of a landscape and what methods to implement further.

Chapter 3: A detailed explanation of the solutions and algorithms used on the framework and problems that have arisen during its development.

Chapter 4: Focus on how the solutions of Chapter 3 were implemented in the framework, following the logic order of execution, since the sketch phase until we obtain the final terrain.

Chapter 5: Presentation of the results obtained and validation of these results based on previously set goals, and understanding if these goals were met.

Chapter 6: Conclusions and final thoughts as well as an overview of future implementations for this work, and way to theoretically implement them.

Chapter 2

Related Work

This section aims to present a revision on the various methods on procedural generation of surfaces, networks and vegetation. A terrain, in order to be realistic, accurate, and give a feeling of an actual natural environment needs to include various components such as mountains, rivers, oceans, different biomes and vegetation. All this can be generated with distinct approaches and techniques. Landmasses will be divided in the following elements:

1. Surfaces (Biomes, Height Map, Erosion)
2. Networks (Roads, Rivers)
3. Vegetation

Each of the following sections will cover each land element and the mechanisms used to accomplish them.

2.1 Surfaces

Surfaces are considered the areas which compose the world land, more specifically the terrain and its characteristics. There are four main solutions for methods of terrain modelling:

1. Stochastic
2. Simulation
3. Sketch
4. Learning

2.1.1 Stochastic Approach

Stochastic methods revolve around using construction rules to give control over the features of the resulting terrain. These rules try to emulate randomness to provide a more natural look to the generated land and are used recursively to improve the level of detail. The randomness factor can, in odd situations, also be a drawback of these methods because it takes control from the developer, as it places the world features in locations that are determined by the algorithms used, and not in the exact position a developer might desire, but unlikely, since the nature of a procedural approach does not contemplate these options.

2.1.1.1 Fractal Methods

The use of fractals to have a base for modelling a terrain was the first approach to the problem. Fractals are geometric figures and Mandelbrot's description of them is "a rough or fragmented geometric shape that can be split into parts, each of which is (at least approximately) a reduced-size copy of the whole" [23]. They are created by applying a certain instruction on an initial figure and repeating the instruction.

Mandelbrot's [24] definition of fractals and Fournier's [11] Midpoint Displacement technique to model a fractal surface, were the foundations for the first works on procedural generation of terrains. Prusinkiewicz and Hammel [36] were the first to produce terrains with different types of topography and characteristics, using fractals and the midpoint displacement algorithm to address the, at the time, problem of generating rivers in fractal generated mountains. The main idea for generating mountains with fractals starts with an horizontal triangle divided in four equal triangles, with each corner having a starter height value. In each iteration, the algorithm adds a new central point (relative to the already existing points in the previous iteration) with a height determined by the sum of the average corner values and a random offset. This offset interval decreases in every iteration based on a parameter that defines the desired terrain roughness. Figure 2.1 shows two mountains, P and Q , generated with the midpoint-displacement method, and the midpoints of the coinciding edges, in order to generate the river, are displaced by vectors of the same length.

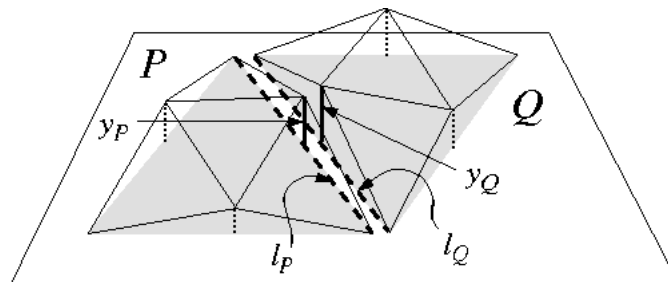


Figure 2.1: Fractal mountain generation representation, where P and Q represent the initial triangles separated by the river. Image from [36], page 2.

Another way to use fractals is using a certain kind called Chaotic Fractals, more commonly known as Noise. Noise has the interesting property of emulating "randomness" which results in height maps with a natural feel and look, compared to real world elevation. Perlin noise [33] is used frequently and consists in creating an initial set of random values to later interpolate with coherent noise. In order to create more level of detail on the terrain, its possible to add various noise layers, each with a specified level of detail, as was demonstrated in [42] by Smelik et al.

However, fractal methods have a major drawback because they can not handle complex structures like overhangs and cave systems.

2.1.1.2 Tiling Methods

Tiling consists in making a division of the terrain in various areas, or "tiles", and using those divisions to assign different values for the parameters on those areas. It is a good way to implement biomes, for example, introducing a "humidity" or "temperature" variable, and assigning different values on different tiles, resulting in different biomes. Voronoi diagrams [45], some examples shown in Figure 2.2, are a well known technique to make this division, as described in [30]. Olsen specifically used Voronoi Diagrams because the noise used to generate the base terrain would keep statistically homogeneous and isotropic, resulting in artificial looking terrains, and the divisions gave the possibility to assign each one with specific parameters called "feature points".

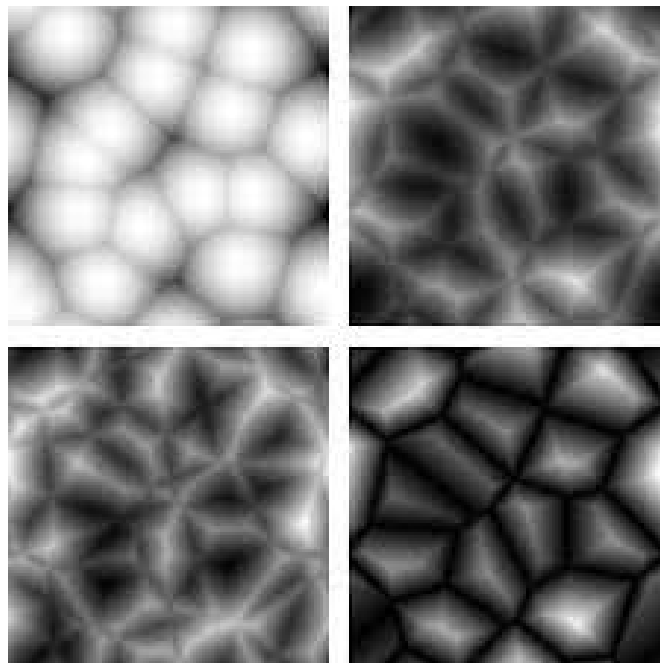


Figure 2.2: Examples of Voronoi Diagrams. Image from [30], page 4.

2.1.1.3 Parametrization Methods

The goal when using these methods is to give a degree of control to the user, to let him generate a terrain that fits his needs, by letting the user control certain parameters, within the generation tool, as shown in [4] by Choros and Topolski. These methods are sometimes used in conjunction with sketch methods (that will be described in more detail later), as shown by [47], where the user sketches terrain features and controls some parameters inherent to those features, like slope and altitude.

2.1.1.4 Grammar-Based Methods

A grammar consists on a set of rules for the creation of formal languages. The first application of this concept on procedural terrain generation was done by Marak et al. [26], where the rules are used to deform terrains and simulate the notion of erosion, in a fractal based terrain. The height of the terrain is represented in the form of a matrix and the rules are, as described by the author, the "context sensitive rewriting of this matrix, representing the erosion process".

In [8] Dey et al. proposes a method using voxel¹ representations of 3D terrains which create features like overhangs (Figure 2.3) and cave systems (Figure 2.4) that cannot be done using a Fractal Method. In fact, the base terrain is generated using Perlin Noise but it uses a semi-automated method for generating features for volumetric terrains using a generation system based on rules. Subsets of a voxel grid are selected to work as the input symbols to a grammar (with operators created by the user), that will generate the terrain features.

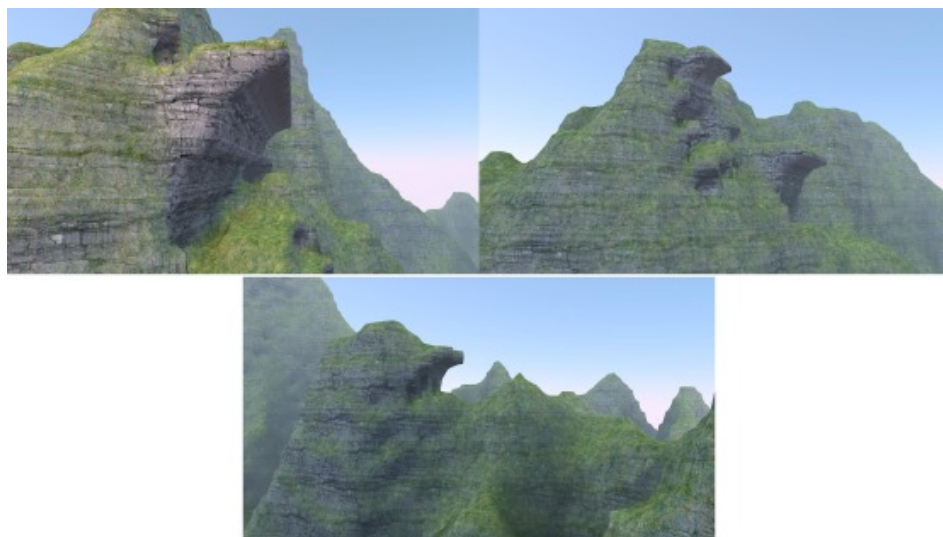


Figure 2.3: Examples of overhangs created using Voxel representations. Image from [8], page 14.

¹Voxels will be later mentioned in greater detail

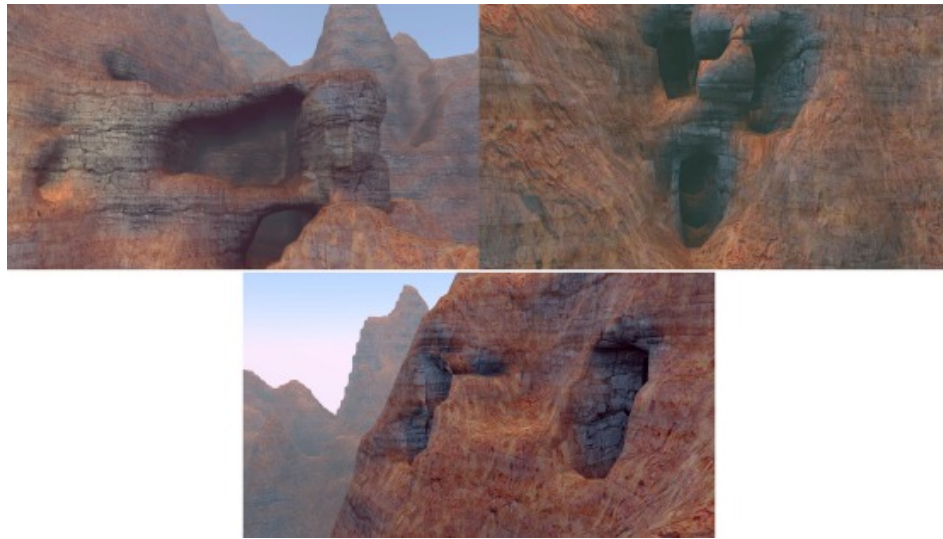


Figure 2.4: Examples of caves created using Voxel representations. Image from [8], page 15.

2.1.2 Simulation Approach

When using simulation methods, the goal is to copy the real world way of carving terrain and build an algorithm that emulates the phenomena.

2.1.2.1 Geomorphological Simulation Methods

The definition of Geomorphology is, according to the British Society of Geomorphology², "(...) the study of landforms, their processes, form and sediments at the surface of the earth (...) includes looking at landscapes to work out how the earth surface processes, such as air, water and ice, can mould a landscape."

As the definition implies, the methods in this section try to simulate erosion produced by the various elements. The methods are applied on a base terrain created usually by fractal methods (or other forms of generation), and carve the land recursively until it reaches a natural look, as shown on the first works done, [17], presented by Kelley et. al, and [28], Musgrave et. al, and later in [40] by Roudier et. al. In [44], Št'ava et. al used a different erosion method. Instead of using fractals to generate a base, a terrain composed of various layers of materials is used and edited by modelling tools that use erosion and deposition algorithms based on real world physics, as shown in Figure 2.5. The model also takes advantage of two hydraulic systems for running water (Figure 2.6). The erosion of certain areas depends on the speed that the water passes: if it is slow, the area becomes less eroded, and more if the water flows rapidly. Also it can implement overhangs that can also fall, if water flows in the base of a cliff. All the materials that get eroded in upper layers later deposit, creating plains.

²<https://www.geomorphology.org.uk/>

In [35] the author proposes a framework to create terrains that can also deal with the problem fractals have with complex structures, as the material is also generated in layers that makes possible to model rocks, arches, caves and protrusions.

Other works like [5], Cordonnier et al. use a combination of the erosion methods already mentioned with tectonics. Tectonics generate crust uplift to generate mountains.

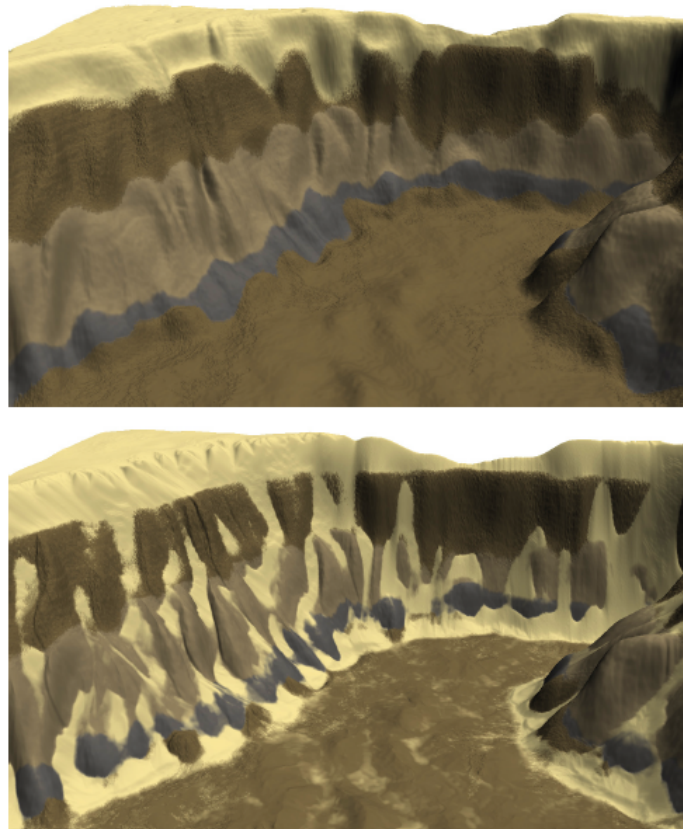


Figure 2.5: Shows all the different layers that compose the terrain, also demonstrating how the top materials deposit over time. Image from [44], page 3.

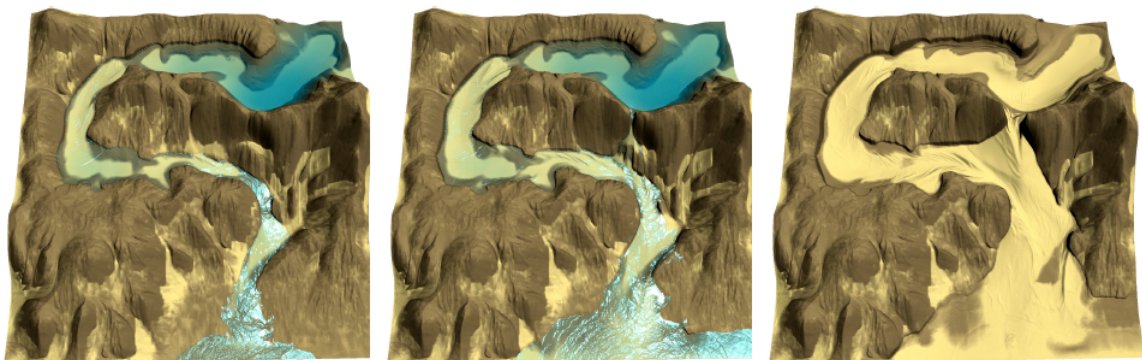


Figure 2.6: Erosion by running water, showing the top materials being dragged over time. Image from [44], page 5.

2.1.2.2 Ecosystem Simulation Methods

The base idea over these methods is that different meteorologic conditions can make two similar looking terrains appear completely different, due to the vegetation a certain climate produces.

In [6], to simulate what composes a real world terrain, Cordonnier et. al proposed that the land was layered, each layer being bedrock, granular material, vegetation and deceased vegetation. The terrain as a whole is affected by physics and environmental effects such as rain, gravity, temperature, wind, fire and lightning. These effects affect the vegetation, producing changes in the terrain, as the author said: "For example, vegetation can prevent rock slides but may equally be destroyed by falling rocks; vegetation absorbs water but flooding may also uproot plants."

A recent approach is [10], in which the authors use a combination of already described techniques and present a final work with various forms of biomes in the generated terrain. The main pipeline for this work starts by generating a base terrain using noise. Next, the biome distribution is computed by using a physics-based approach to make a climate simulation using parameters like temperature, wind and rain. The third pipeline step aims for a terrain refinement, using Digital Elevation Models to obtain realistic biome-specific terrain details. Finally, the last step is asset placement to populate the terrain with assets like trees, bushes, vegetation, and others, and make the world richer, with more detail. Figure 2.7 shows the final representation of various biomes.

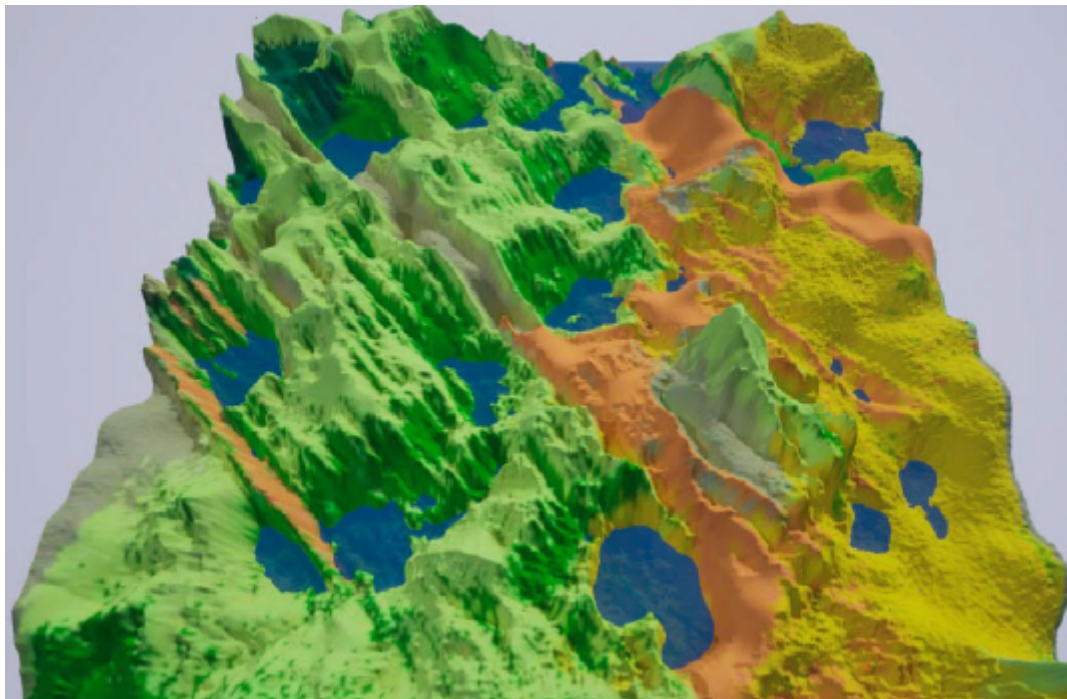


Figure 2.7: Biomes are depicted using colours and terrain characteristics are adapted to the respective biome. Image from [10], page 6.

2.1.2.3 Agent-Based Simulation Methods

Although Doran and Parberry in [9] states "(...) there currently is no consensus on what a software agent is (...)", he starts with the definition proposed in [41] by Russell and Norvig: "(...) an agent perceives its environment through sensors and acts on it through effectors." So the agent, after perceiving the environment, affects it by destroying or building. The agents' phases are: coastline, landform and erosion. The first phase creates an outline for the land, usually surrounded by water. The landform phase sculpts the terrain, like mountains, plains, beaches and others. The last phase erodes these features creating rivers and making them realistic.

2.1.3 Learning Approach

Learning methods use real-world data and information (in form of images or Digital Elevation Models) and create terrains with the features it learns.

2.1.3.1 Example Methods

These methods extract terrain features from a set of examples. In [48], the Zhou et. al describe a technique that uses a terrain example model and lets the user sketch terrain features. The algorithm extracts the features from the example terrain and combine it with the sketch, as shown in Figure 2.8.

In [1] the authors create a dictionary extracted from a set of examples, as pre-processing. The dictionary analyses multi-layer input to create multi-layer atoms (which include information on elevation, vegetation, light and water flow). The dictionary is actually a set of two dictionaries, low and high resolution, with the same number of atoms. This way, a reconstruction can be obtained from the low resolution dictionary by replacing the atoms in a one-to-one correspondence. The input terrain is low resolution and contains elevation layers. It can be user sketched or a real digital elevation map. The atoms are extracted by decomposing the original terrain into patches and extracting them. New atoms are created if when decomposing a patch, there are no two atoms that combined produce the first atom. When finally combining the atoms, a new terrain is created.

A very recent approach to these methods is using Generative Adversarial Networks³. In [31] Panagiotou and Charou propose a method to generate realistic terrains based on satellite and drone footage of landscapes. This method consists in two major steps: in the first one, a GAN trained with all the gathered images learns the distribution of the data set and creates new satellite images. In the second step, the image needs to be converted from RGB to a Digital Elevation Model, and a Conditional Generative Adversarial Network is used for it. In the end,

³GANs were originally introduced in [13] and consist in a class of machine learning frameworks. Essentially, GANs learn to generate new data with the same statistics as the training set.

the RGB image and the height map generated are combined to create the final terrain.

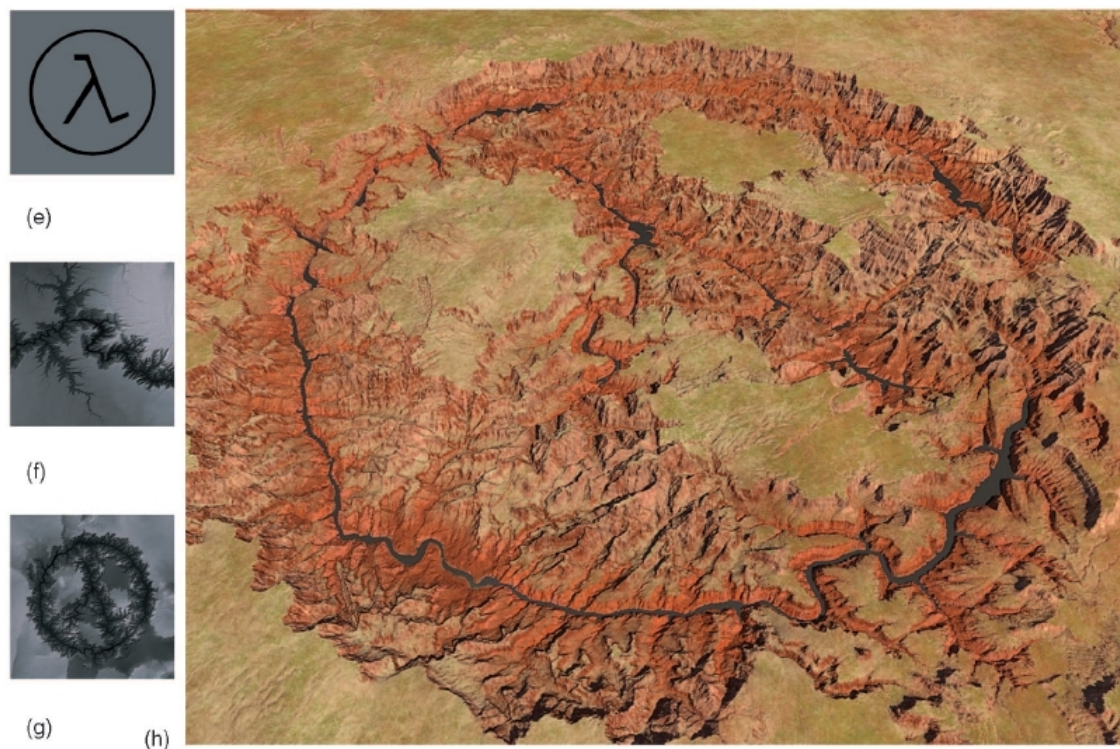


Figure 2.8: (e) Sketch. (f) Grand Canyon DEM. (g) Terrain Synthesis. (h) Final terrain render. Image from [48], page 8.

2.1.3.2 Search-Based Methods

Search-Based Methods are usually centered around evolutionary algorithms, which evaluates from a set of examples with a fitness function. In [46] the authors introduce a system that interacts with the user in a web-based User Interface that lets the user choose preferred terrain settings and then the genetic algorithm uses the user preferences to explore a multi-dimensional parameter space to satisfy user's wishes.

2.1.4 Sketch Approach

Sketch methods generally take as input a sketch done by the user. This sketch usually represents terrain features the user wants to see in the generated world and an algorithm takes the sketch and applies the features described in it to the final product.

2.1.4.1 2D Methods

In [47] Yin and Zheng propose a method where the user draws plains, mountains, lakes and other terrain features using colours. The algorithm interprets them and determines a standard height for each zone, and a terrain skeleton is built.

In the case of [25], Mangra et. al propose a way for the user to draw a sketch of an height map and the algorithm populates the terrain with the features associated with the input height it reads from the sketch.

2.1.4.2 3D Methods

Using a 3D Sketch approach the user needs to draw the sketch using 3D vectors. In [2] the user draws in 3D using curves that are interpreted by the algorithm to create complex structures as shown in Figure 2.9.

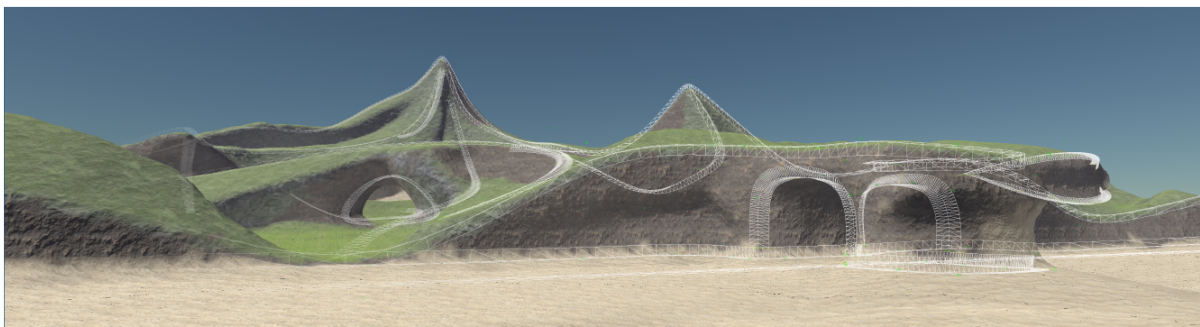


Figure 2.9: The curves used in the sketch are represented in white and model the terrain features. Image from [2], page 2.

2.2 Vegetation

Vegetation and plants was the first procedural content to be generated, as it was the base for the theoretical knowledge related to L-Systems.

2.2.1 L-Systems

L-Systems⁴ were first mentioned in 1968 in [21] by Aristid Lindenmayer, who was a botanist. He was interested in modeling the way plants evolve and developed a system to achieve it. The base for Lindenmayer's work was the Morphogenetic Theory, [20], where the authors describe the way multi-cellular organisms evolve, explaining how cells interact and acquire new

⁴Acronym for Lindenmayer-System

properties they would not have with their individual capabilities. A good example for that is the way a seed can, with time, evolve to a fully grown tree.

Lindenmayer created a formal grammar. He used a set of characters, each describing a part of the plant. Production rules are then used to change the string of characters. So, using the initial set and applying rules iteratively can model the development that occurs in plants.

Later, in [43], Smith realized the potential of L-Systems for data expansion starting from a simple set of initial data and the investigation on this subject, in the area of Computer Graphics, was generation of plants.

2.2.2 Generation Methods

Following the work done in [43], Lindenmayer and Prusinkiewicz published a book, [37], that would later become the most important reference for the application of L-Systems in Computer Graphics, because before it, the works regarding the subject were using Fractal Methods which were not producing realistic looking results.

In [16] Hammel et. al propose a method for generating plants procedurally, making use of Lindenmayer and Prusinkiewicz work, using L-Systems. He starts by creating a model from the observations of the plant's growth and makes the necessary measurements. Finally, functions that describe the plant's growth are defined from the data statistical analysis.

Another aspect that was interesting, was the generation, not only of the plant, but all the stages it went since the beginning, to create an animation of a determined plant's life, using a sequence of images. In [38] Prusinkiewicz et. al accomplish this using differential L-Systems.

In [27] the authors created a method of generating other forms of plants besides singular plants (like trees or flowers). His method handles the generation of clonal plants, roots, bindweed and the competition between trees for light (resulting in a bigger and stronger tree from the one who took more light during the growing phase).

Motivated by the film industry, the development of complete natural environments was becoming a matter of interest in the late 90's and early 00's. In [7] Deussen et. al describe a system uniquely for the visualization of plants in a complete ecosystem. The base terrain is generated from some method, using real life data (or not). Next, an initial plant distribution is done in the terrain, again, using real data or specifically placed, and finally the algorithm, based on the defined ecological model, generates a new distribution for the plants, adding new plant models, selected at random from a set of similar plants on a given location, to the terrain.

In recent years, with the advance of technology a new demand is rising, namely real-time procedural generation. In [39] Roque and Atienza propose a method for generating the growth, in real time, of corn, taking into consideration the various parts of the plant, leaf lamina, leaf sheath and stem. The growth is controlled by parameterized growth functions that produce

different results depending on external factors such as nutrient uptake. Leaf curvature and undulation are determined using a spring-based rigidity model attached to the leaf's midrib that is affected by gravity. Figure 2.10 shows the various life stages of the corn plant, growing over time.



Figure 2.10: Corn plant life stages. Image from [39], page 1.

2.3 Networks

In this section we will be covering Networks. In a terrain, a network will be considered any element that can be represented in the form of a graph or path. The two main elements with this characteristic are rivers and roads, so methods for generating these structures will be revised in the next sections.

2.3.1 Roads

One of the first works regarding generation of roads, by Parish and Müller, is [32] which uses L-Systems. The method follows the same theory as generating plants: starting from a first road segment, every other road is generated from the first one iteratively, following construction rules that are chosen randomly. However, some problems rise because the algorithm may generate roads without a way out, or meaningless intersections. To correct this, we need to add some restrictions in the generation process. This is called a Self-Sensitive L-System. In [18], Kelly and McCabe propose an application called *CityGen* that essentially works using the process that was previously mentioned, but the user can specify the location for the main intersections before hand. This will create primary and secondary roads and the algorithm will adjust the roads accordingly, in order to create a reasonable territory ordering, inside the city.

More complex works take into consideration notions like land use⁵ and districts⁶. They also approach the subject of the pattern roads in a city may have: radial, raster and population-based. Groenewegen et. al, in [14] explore these notions thoroughly, as they compare North American and European cities and the way they evolve through time, and the result of said evolution in the city roads. Before generating the city, parameters must be decided, the diameter of the city, the continent on which the city is located (in order to decide the distribution of the districts), historical background (to determine the districts in the oldest part of the city) and finally the number of highways passing through the city. Figure 2.11 shows the evolution phases to generate a new city, starting from the terrain and main roads, and last the streets, based on the surrounding districts and locations.

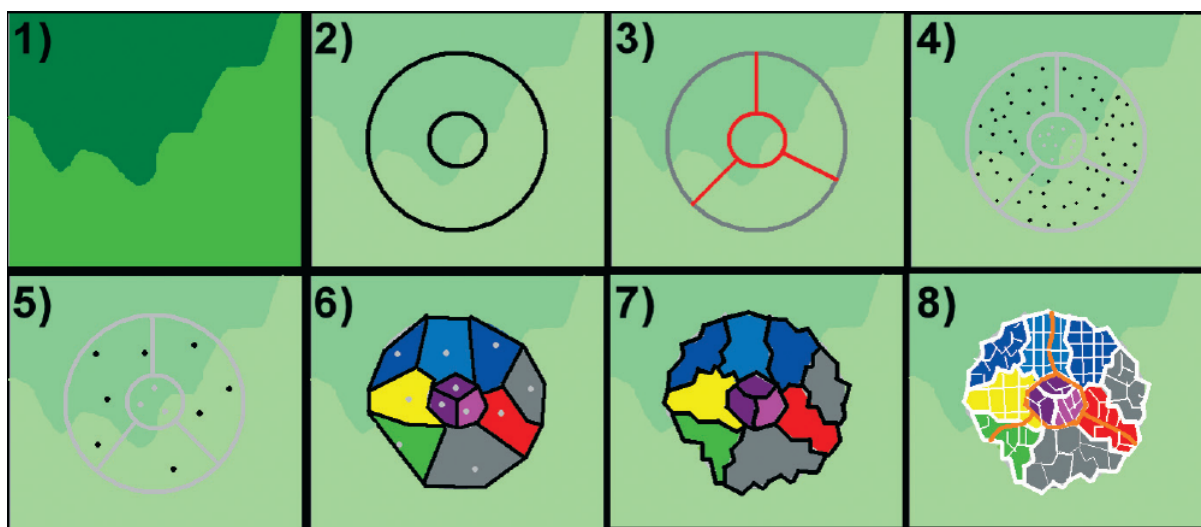


Figure 2.11: Phases used to generate the city: (1) Terrain. (2) City Limits. (3) Preliminary Highways. (4) Candidate Locations. (5) District Locations. (6) Voronoi Graph. (7) Noise. (8) Completed city. Image from [14], page 3.

A more recent approach, [19], Kelvin and Anand use Conditional Generative Adversarial Networks to generate roads to use in game development by sketching primary roads in a graphical user interface. The interface then outputs a road network based on the sketch and adds secondary roads.

2.3.2 Rivers

Generation of rivers differs from road generation. As in real life, a river needs a source and a mouth, and most Procedural Generation algorithms for generating them, use both elements. In river generation algorithms, usually either one is chosen to start the generation.

⁵Purpose to which a section of land is used for: commercial, residential, industrial and others.

⁶Division in economic power inside a city, to divide residential areas, for example.

To generate a river starting from a source, generally it is used a method with some filters in order to choose a high location without choosing every possible high point and end up with an unrealistic amount of rivers. From here, one can use a fast search algorithm to search for the next nearest lower point until it reaches the ocean. This produces a fast, but not so realistic result. Another way is to use a physics simulation engine. In [3] Belhadj and Audibert, upon selecting the desired sources, uses a physics based model to make water particles that are spawned in the source location and get affected by gravity, and, associated with a friction coefficient, carve a river bed, as shown in Figure 2.12.

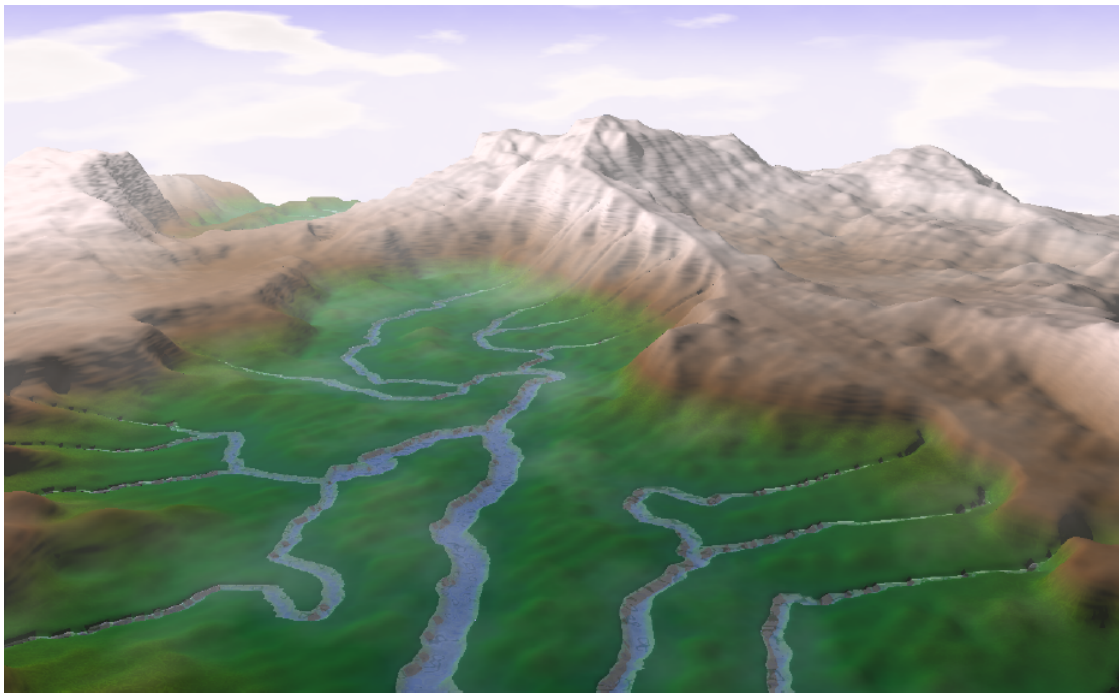


Figure 2.12: Final result of generated river network. Image from [3], page 4.

When the river mouth is chosen for starting point, logic is reversed. In [12] G  nevaux et. al follow this line of thought. Instead of rivers flowing down the mountain and joining when getting close to the ocean, the river flows up and splits, creating creeks, until it finds the highest location. In the end, it creates complex and realistic river networks.

2.4 Conclusion

The intention of this section was to demonstrate different alternative methods and approaches to generate landscapes, as well as networks and vegetation in order to understand what are the best choices we can make for the framework we want to develop. Although some of the work we showed is tightly related to the thesis project we want develop, namely the sketch and user related works, none try to accomplish the same as we want.

The goal of the framework is a terrain generator that uses a sketch to create the surface, but in a different way of the terrain generators demonstrated during this section, as they usually tend to create terrain features from a sketch, and not the actual landscape, in terms of its ordering and topography.

The networking section was included as a way to also get inspiration on methods and approaches to include generation of roads (by having a robust and coherent networking system for roads, we create a base to include coherent generation of cities, since one of the main issues with generating cities is precisely the difficulty of ordering the terrain in a realistic way).

The next chapter will give an overview of the developed framework.

Chapter 3

Framework Overview

This chapter presents the decisions, approaches, algorithms and tools used regarding the implementation of the terrain generation framework. We will break it down into each relevant component and explain the idea behind of what is trying to be achieved. The goal of this framework, as said before, is for it to generate finite square terrains, where the territorial ordering is completely decided by the user. The user decides this ordering, painting on a blank canvas using colours, where each colour will represent a biome/region, with all the terrain features associated with it. The user will also have the possibility to generate different terrains using the same sketch, by altering the starting *seed*, until the preferred one is found.

3.1 Architecture Overview

The framework was developed on top of Unity Engine, using its API and capabilities, due to the widespread usage throughout the Computer Graphics industry and all the features it offers out-of-the-box. The main idea behind this framework is to give a user full control on the output, which is the generated terrain, in terms of size, terrain features and textures. Usually, terrain generation software does not give this degree of terrain customization, because it usually follows strict generation rules implemented on the algorithms, giving less options to the user, like the terrain ordering and how the general terrain looks, regarding the texturing used. The main goal is to generate the final landscape, but leave the framework open to add any new components to the existing ones: the addition of any new number of biomes, new terrain features inside of each biome to better represent that region's essence, possibility of changing the textures used to give the look the user wants, addition of new mechanics, or any other expansion one finds suitable, so that the framework can be used as a base for different applications, like video games, for example. All the work involving textures and tile maps was developed in Photoshop.

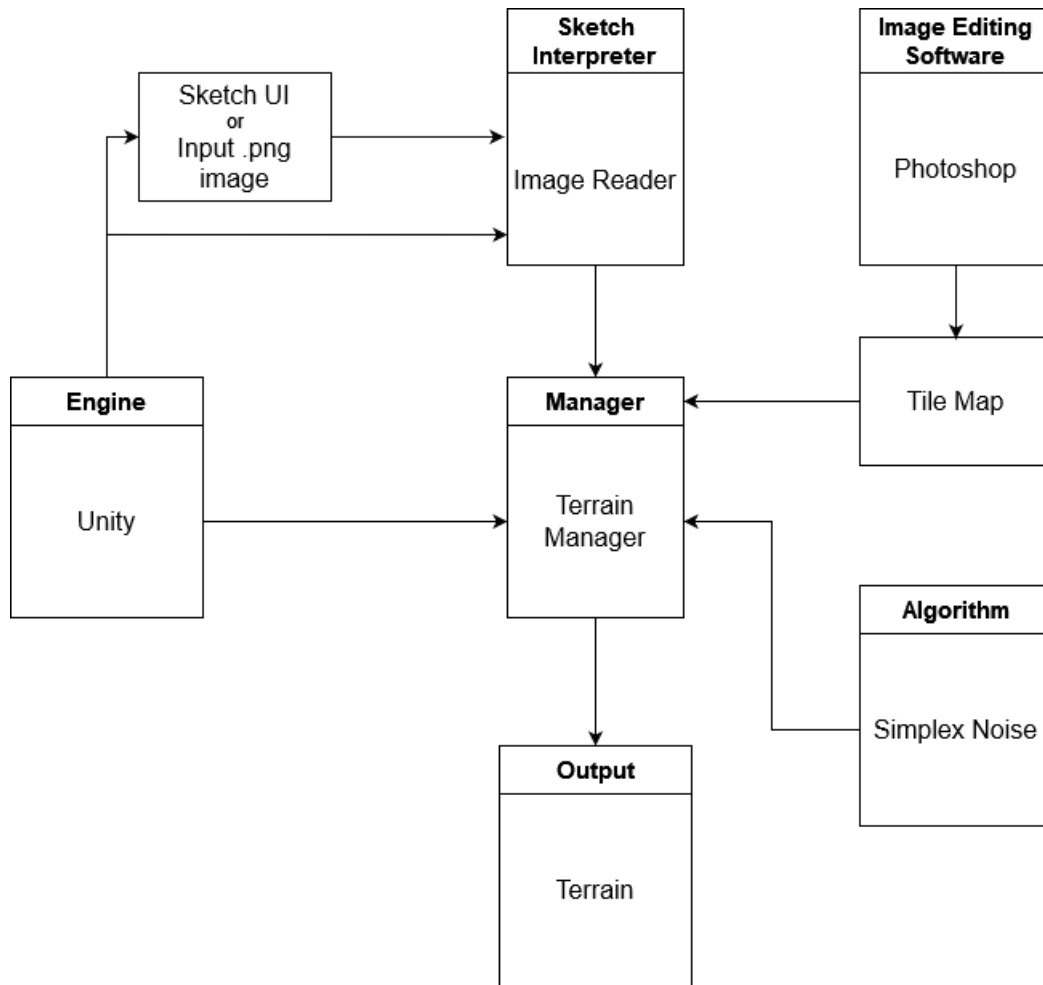


Figure 3.1: Unity Engine is used to build a Sketch UI, an interpreter for the sketches and a Manager that uses the Simplex Noise algorithm and a tile map built and edited in Photoshop to generate a terrain.

3.2 Framework Features

The features that integrate this work will now be demonstrated, following the logic order of the framework usage.

3.2.1 User Sketch

When we start using the framework, before the generation phase, the user needs to draw a 2D sketch, using various solid colours, in a canvas with size $[w * h]$, where w represents the total terrain width along the x -axis and h the terrain height along the y -axis, so $[w * h]$ is the terrain size seen from above, where each colour will represent a biome. A biome in this work's context is an area in the terrain which follows the same premises regarding topography and

terrain features. The concept of biome will be explained in greater detail in the next sections. The sketch can be made inside the framework with the help of an UI, and the user will also have the opportunity to choose between a selection of predefined terrain sizes, before colouring it. The available terrain sizes in the UI are $[32 * 32]$, $[64 * 64]$, $[128 * 128]$, $[256 * 256]$, $[512 * 512]$, $[1024 * 1024]$.

To colour the sketch, the user will have the possibility to choose between 8 possible colours:

- Blue
- Yellow
- White
- Green
- Black
- Cyan
- Magenta
- Red

The canvas is white before the user starts to paint it, but since the colour white also represents a biome, the user needs to make sure he paints all the pixels he wants correctly, as leaving white spaces between other colours will generate the biome associated with white, in that respective position. The next image, 3.2 will show an example of a sketch using all the colours available, that will generate a terrain with the size $[256 * 256]$.

Besides giving the user the option to draw the sketch in a dedicated interface, we will also provide a way for the user to simply input a sketch he made, as long as the sketch respects the intended size the terrains must have. In this case, all terrains, and therefore sketches, need to be a square where the side size is a power of 2 greater than 32.

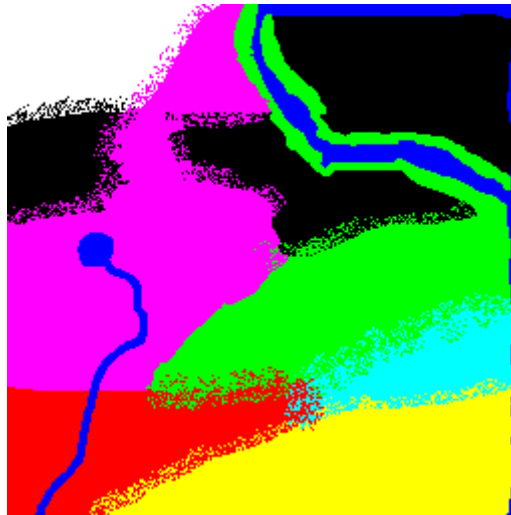


Figure 3.2: Example of a sketch a user can do.

3.2.2 Voxel

The word **Voxel** was created analogously to the word "*Pixel*" and according to its etymology, "*Vo*" represents *volume*. In a way, a voxel acts like a 3D pixel. It represents a section of a 3D grid, and in this framework, a voxel represents the most basic terrain unit, with the size $[1 * 1 * 1]$ and will be called **block**.

The most common block used in the framework for generating the terrain is the **Stone Block** because it is the only one present across all the different biomes. It is the unit used to shape all the terrain topography, and all other biome-related blocks, are placed on top of Stone.



Figure 3.3: Example of a voxel used in the framework, the Stone Block.

3.2.3 Chunk

A chunk is a term often used when working with voxel based terrain generation, as it represents a terrain parcel. It is usually inefficient to generate a complete terrain (depending on the size) in a single parcel. Also, if the terrain is used in a video game, for example, when dealing with memory issues, it is more useful to destroy and create chunks around the player as necessary, instead of having them all loaded at the same time, which can result in a memory shortage, and this is the main reason we are going to use chunks to generate the terrain, even though it will be displayed as whole all the times. Having the basis of the generation phase being done using chunks, facilitates the implementation of displaying the desired chunks dynamically in a later stage, when memory concern is more relevant.

Regarding the framework, a **chunk** is a cubic area with the dimension $[cW * cH * cV]$, where cW is the chunk width, cH is the the horizontal chunk height and cV represents the vertical height of the chunk, the difference between the highest and lowest block. The size of the chunk was decided to always be the same (it can be modified though), but the number of chunks instantiated depends on the terrain size. For example, if the user chose a canvas with a size $[32 * 32]$ pixels, and $cW = cH = cV = 16$, the terrain will be composed of 4 $[16 * 16 * 16]$ chunks. Since $[32 * 32]$ will be the minimum terrain size, we will use $[16 * 16 * 16]$ chunks for all the terrain sizes, since it also works well for larger terrains. It would make sense to include smaller chunks if the framework included the possibility to generate smaller then $[32 * 32]$ terrains, but worlds this small will not accurately show terrain features, and so, they become redundant. With this in mind, and as said before, the project will only be able to generate terrains where the side length is a power of 2, greater than $2^5 = 32$.

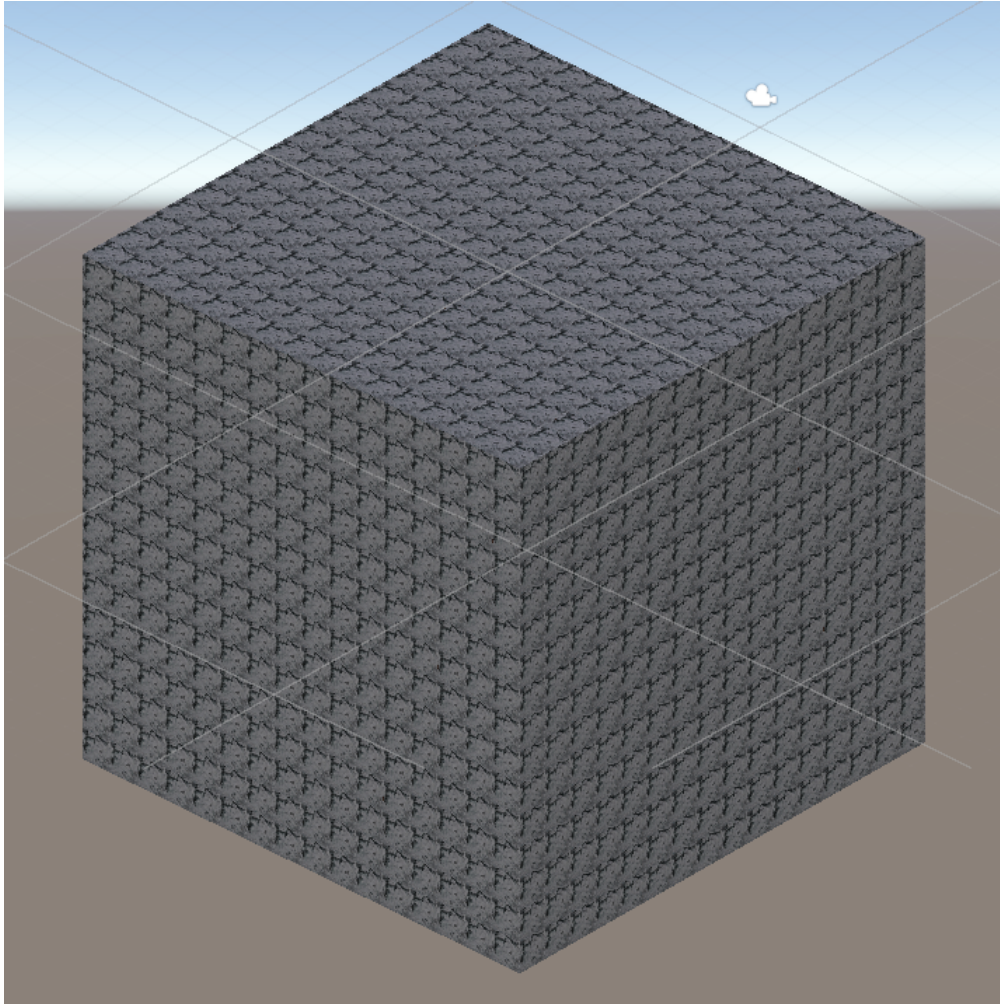


Figure 3.4: One chunk made of Stone Blocks with the size $[16 * 16 * 16]$.

3.2.4 Terrain

A terrain consists on a square 3D grid composed of chunks with a fixed size. Depending on how the user painted the canvas, the resulting terrain will be composed of one or more biomes, each of them with different characteristics. The next section, will define biome in this framework context, present all the possible options the user can choose and differentiate them based on topography and vegetation.

3.2.4.1 Biomes

According to the National Geographic Encyclopedia¹, "A biome is a large area characterized by its vegetation, soil, climate, and wildlife.". The biomes that exist in planet Earth can be divided in five major types that are further divided into more specific categories. The major biomes are:

¹<https://www.nationalgeographic.org/encyclopedia/biome/>

- Aquatic
- Grassland
- Forest
- Desert
- Tundra

Aquatic biomes can be divided in freshwater and marine. Freshwater being bodies of water surrounded by land, like rivers and lakes which have a low percentage of salt. Marine biomes are the largest areas on the planet, high percentage of salt, and can be oceans, coral reefs or estuaries.

Grasslands are open areas mainly composed of ground vegetation like grass and small bushes, which result in a dry and warmer climate. The two major grassland biomes around the world are tropical grasslands (also known as savannahs) and temperate grasslands, which are prairies and steppes.

Forests are areas mainly composed by trees. The three major forest types are temperate forests (which are usually at higher altitudes and experience all the four seasons climates), tropical forests (hot, high on humidity) and boreal forests, also know as taiga (higher altitudes than temperate forests, resulting in dry and cold climates).

Deserts are mainly dry areas, that have low biodiversity. Hot and arid, semiarid, coastal and cold are the types of deserts that exist.

Finally we have tundras, which, according to the National Geographic Encyclopedia, have "(...) extremely inhospitable conditions, with the lowest measured temperatures of any of the five major biomes (...)". The two types of tundra are arctic and alpine.

The characteristics of each biome are completely dependant of one rationale factor: the actual position they have in planet Earth. In this framework's context, this factor is irrelevant because not only we are creating an arbitrary square of terrain, with no connection to Earth, but the user has the power to order the terrain has he wants. If the user wants to place a dry and arid desert right next to a Taiga, nothing stops him, so its in the hands of the user to create a realistically ordered terrain, following Earth's logic, if he desires to do so. Another important aspect, is how these biomes blend with each other. The sketch dictates the outcome of the terrain, and a user may want the biomes to be blended together on their borders. The framework leaves this detail in the hands of the user, the way he wants the biomes to interact in the borders is his choice. Figure 3.2 demonstrates how one can blend borders in the sketch phase, by mixing the colours of each biome in borders. The results of blending the borders can be noticed on the Results and Validation chapter.

Regarding the framework, we are not including every biome that exists on planet Earth, as it is not viable to do so, mainly because of the size of each terrain we are generating and the

characteristics each one has, which can be too similar. Giving the user the possibility to add or alter any biomes he wants is more interesting, since the biomes we created are used to give a sense of variety, and adding all possible biomes to this work would not provide much value. Also, for this thesis, different biomes can be compromised into a single colour. Water, for example, can be an ocean, a river or a lake, depending on how the user draws it on the sketch. Other biomes have vegetation, but since we provide the option to turn off the generation of vegetation for each biome who has it, biomes like a desert, if it does not have vegetation, can be used as a beach. For this reason, we are committing a linguistic inaccuracy when calling each coloured area, "biomes", region being the most appropriate term. We will disregard this Biology/Geology error, and keep calling each region a biome, because this term better represents what we want to achieve with the different colours. So, the "biomes" that are included are:

1. Water Masses (That can be either lakes, oceans or rivers)
2. Hot and arid Desert
3. Cold Desert
4. Temperate Forest
5. Taiga
6. Tropical Forest
7. Alpine Tundra
8. Savannah

Each of these biomes have specific features, such as vegetation, wildlife, height on which they occur and the main soil constitution, all of these components will be discussed further and an image of a real life biome will be presented, to show what is trying to be accomplished.

1. Water Masses



Figure 3.5: Water Mass. Image from <https://www.boatingpartnerships.com.au/>.

Vegetation and Soil Materials

Water masses will not have specific vegetation, as they act only as simple water bodies, and because of the ambiguity they represent, being either oceans, lakes or rivers. The soil, under the **Water Blocks**, is made of stone. The main idea with this biome is to give a sense of water on each determined location, so the soil it uses is not relevant.

Topography and Cave Systems

Terrain elevation in water regions is not relevant because it is done solely so that the water is completely flat at the surface and at a low height, relative to the other biomes. This biome will not have cave systems, as they would not pose any significance for the purpose of this biome.

2. Hot and Arid Desert

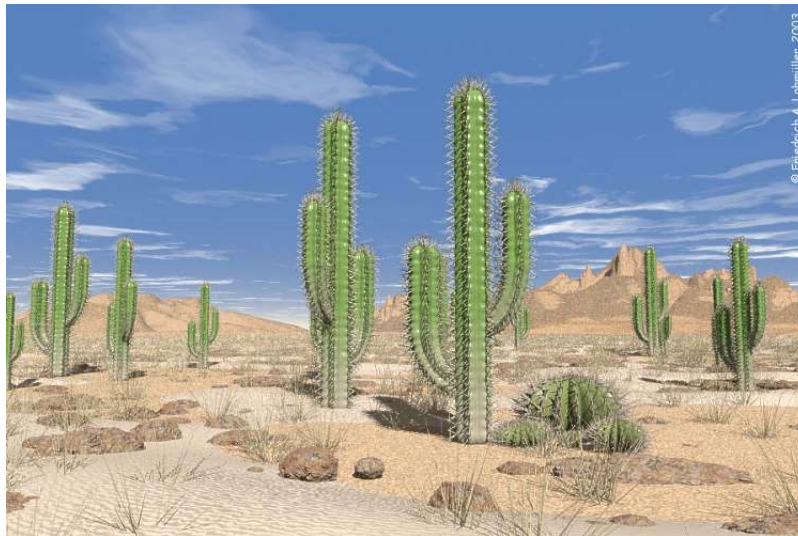


Figure 3.6: Hot and Arid Desert. Image from <https://edu.glogster.com/glog/cacti/2vf33wnklhk>.

Vegetation and Soil Materials

Deserts², although they seem harsh environments for plants, they have diverse, but sparse vegetation. The most common plant that can appear in deserts are cacti, with various sizes, as well as small bushes. Usually composed mainly of sand, and for that reason the main block in this biome is the **Sand Block**.

Topography and Cave Systems

Deserts are generally flat, excluding dunes and undulations on the sand. For that reason these biomes will have low elevations and depressions on the terrain, only to emulate the way sand is distributed. There will be sporadic entrances to cave systems, but most are covered with sand, and only the larger entrances will be visible.

²According to the USGS: <https://pubs.usgs.gov/gip/deserts/features/>

3. Cold Desert



Figure 3.7: Cold Desert. Image from <https://www.newscientist.com/article/mg20527426-200-chilling-out-in-the-coldest-place-on-earth/>.

Vegetation and Soil Materials

Exactly as the hot desert, the cold one also has harsh conditions for any kind of vegetation to thrive. Many cold deserts do not actually have any kind of vegetation, due to the amounts of ice and snow that populate those regions. However, some small resistant bushes and plants may survive under these conditions. The soil is completely made of snow and ice, for that reason the main block is the **Snow Block**.

Topography and Cave Systems

Cold deserts will be, for the most part, completely flat, and like the hot deserts, only some ripples and undulations will be added to the terrain. Also like the hot desert, most cave system entrances are covered with ice and snow and only the bigger ones will stand out.

4. Temperate Forest



Figure 3.8: Temperate Forest. Image from <https://wattson.pt/2020/10/18/mecanismos-de-alerta-precoce-devem-ser-reforcados-para-prevenir-incendios-florestais/>.

Vegetation and Soil Materials

Temperate forests thrive with vegetation, because they have a very broad climate, that usually experiences the four seasons. It has some ground vegetation like bushes and herbs. As trees in this biome tend to be deciduous and have broad leaf, the oak was the tree chosen to populate it. The soil is usually composed of fertile dirt.

Topography and Cave Systems

Temperate forests do not have significant elevations and are mostly flat. Nevertheless, small dens and cave entrances can be found scattered across the terrain.

5. Taiga



Figure 3.9: Taiga. Image from <https://www.infoescola.com/biomas/taiga/>.

Vegetation and Soil Materials

Although not as harsh as a Cold Desert, the Taiga has a really cold climate, where in taller locations usually snows. The trees in these forests are evergreen, highly resistant to cold weather and ice. The tree that is represented in this biome is the Pine. Soil is not too different from a temperate forest and is composed of dirt.

Topography and Cave Systems

Taiga biomes are quite irregular and hilly and can have mountains with sufficient height to snow. Overhangs can be seen in some places, due to the rough terrain. Caves are common and usually large.

6. Tropical Forest



Figure 3.10: Tropical Forest. Image from <https://www.mo.be/analyse/de-staat-van-onze-regenwouden-11-zaken-om-op-te-volgen-2021>.

Vegetation and Soil Materials

Tropical biomes are very rich and packed with vegetation, producing very dense forests with trees, bushes and heavy ground vegetation. The climate is very favourable for plants to thrive, as it is hot and humid. The tree species that will mainly be represented in this biome, is the ceiba. Grounds are made of rich dirt.

Topography and Cave Systems

Has in a normal temperate forest, the terrain in a tropical biome is mostly flat, excluding some elevations and depressions on the terrain. Small dens and holes are possible to be found but the heavy vegetation usually hides these formations.

7. Alpine Tundra



Figure 3.11: Alpine Tundra. Image from <https://www.colorado.edu/today/2019/03/21/alpine-tundra-releases-long-frozen-co2>.

Vegetation and Soil Materials

The Alpine Tundra is a very harsh and icy environment that does not allow trees or much ground vegetation to grow, except for some grass patches and small bushes. The soil is composed of dirt and stone for the most part and snow in taller locations.

Topography and Cave Systems

Tall mountains can be observed in this terrain, and due to the absence of vegetation, all formations of any sort of cave system, are usually completely exposed and common.

8. Savannah



Figure 3.12: Savannah. Image from <https://tmarndt.wordpress.com/author/tmarndtwccc/>.

Vegetation and Soil Materials

The hot and dry climate these regions have, results in a very low number of trees, as they have to compete for water sources. The intense sun prevents smaller vegetation from growing and dries the little grass that manages to grow from the dry dirt.

Topography and Cave Systems

Usually completely flat, topography in a Savannah is similar to the temperate or tropical forest. Small caves and dens can be found, although not very common.

3.2.4.2 Tile Map

A tile map is an image created in an image editing software that contains all the textures used to generate our terrain. Figure 3.13 shows the tile map we are using for this framework.

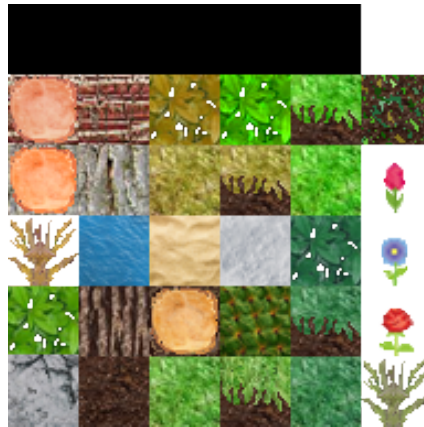


Figure 3.13: Tile Map

All the textures were built using real world images of what we were trying to represent, but if this framework is used to generate a terrain that is not supposed to look realistic, the tile map can easily be swapped by another that is more adequate for the goal the user intends.

3.3 Noise

When generating genuine and lifelike formations, objects or motions, we need to choose random values from a set that we can evaluate and validate as natural, beforehand. It is not wise to choose genuinely random values, because randomness usually results in chaos, and although "nature" can sometimes seem random, when dealing with the generation of "nature", it is safer to decide what is expected natural to look, and make our randomness based on those expectations. This makes the output pseudo-random. With this in mind, a noise function was used to introduce randomness to this framework, and it is used to generate everything that needs to be included in the terrain and needs to appear natural: terrain topography, distribution of vegetation and generation of caves. The two following sections will present two forms of noise, the first one being the predecessor of the second, which is the one used in this framework. An explanation will be given about a form of noise that we are not using, simply because they both consist on the same basis, and is more convenient to approach the concept as a whole.

3.3.1 Perlin Noise

Perlin Noise was developed by Ken Perlin and was presented in [33] with the goal of generating natural looking formations. Figure 3.14 shows a 2D gray scale representation of what Perlin Noise looks like.

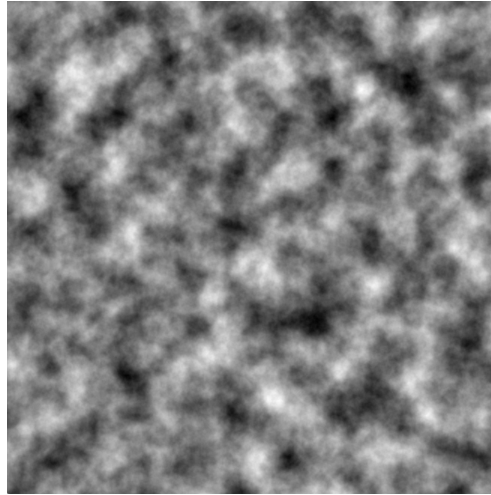


Figure 3.14: Perlin Noise. Image from <https://medium.com/@yvanscher/playing-with-perlin-noise-generating-realistic-archipelagos-b59f004d8401>.

A Perlin Noise function needs:

- Noise Function
- Interpolation Function

A noise function is essentially a seeded random number generator: if an integer is used as an input, the function returns a random number based on that parameter. When the same input is used twice, it will output the same random value, this is why we call the input the "seed", because a certain *seed* will always originate the same outcome. The figure shows a simple noise function, that for every value on the x - *axis*, a random value between 0 and 1 is assigned:

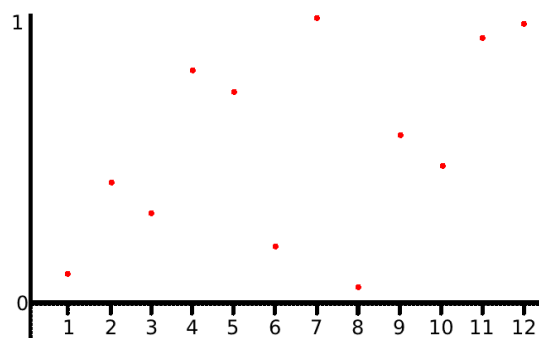


Figure 3.15: Discrete Noise Function Graph

By interpolating these discrete values, we define a continuous function. Now, we can obtain random numbers from non-integer numbers:

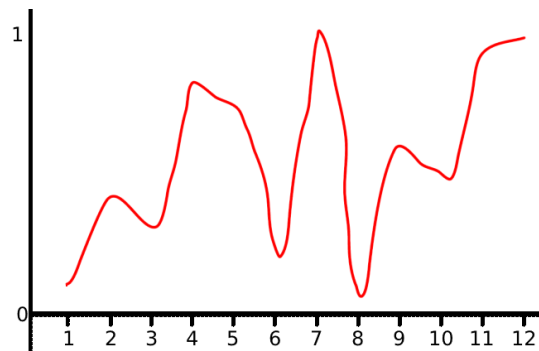


Figure 3.16: Continuous Noise Function Graph

In order to achieve a Perlin Noise function we add together different continuous noise functions, each with their respective **amplitude** and **frequency**³, as shown on Figure 3.17.

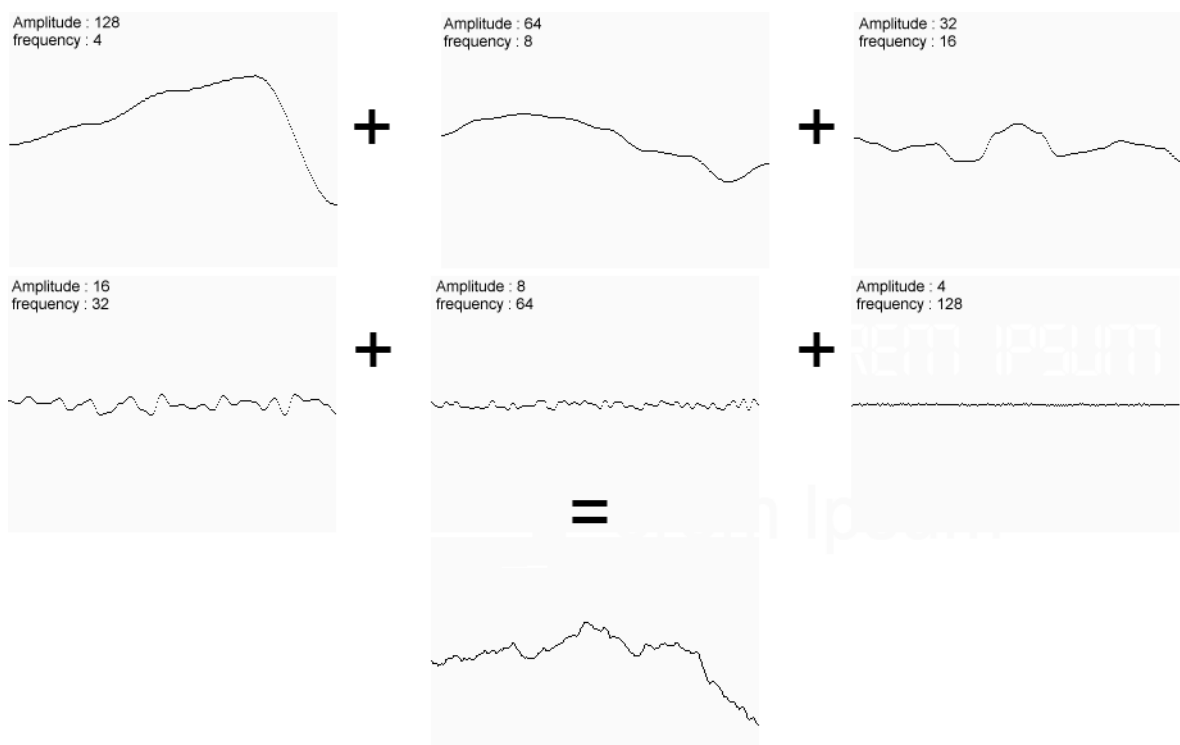


Figure 3.17: Result of the sum of 6 noise functions, each with double the frequency and half the amplitude of the previous one. Image adapted from <https://adrianb.io/2014/08/09/perlinnoise.html>

³Since we are working with waves, we should define both these concepts. The **wavelength** of a certain wave is the distance from one peak to another. The **amplitude** is the height of the wave. The **frequency** we define as $1/\text{wavelength}$.

When adding different noise functions to achieve Perlin Noise, one may wonder what frequency and amplitude values are more convenient to use. Usually, Perlin Noise is created with noise functions that have twice the frequency and half the amplitude for each consecutive step. This setting creates the classic Perlin Noise, but it can be altered to accommodate any change necessary to the output noise. To deal with both these parameters at the same time, a measurement that specifies the amplitude of each frequency is used, called **Persistence**. Essentially, Persistence is a value between 0 and 1 that determines how much each noise function contributes to the final noise value, so it essentially controls the amplitude.

Each successive noise function we are adding is called an **Octave**. One can add any number of octaves he wants to achieve Perlin Noise, but if we add too many, there will come a point when an octave has too high of a frequency to be displayed, because each successive octave has twice the frequency of the last one, and a screen can only show detail until a certain point before it runs out of pixels. Regarding Persistence, if it is equal to 1, all octaves contribute equally. Persistence can actually be greater than 1, which makes successive octaves contribute more, which essentially tends to produce regular static (Figure 3.18), and does not produce the results that we consider realistic, as the next figure shows. This is the reason it is advantageous to set Persistence to a value that is less than 1. Figure 3.19 shows how different Persistence values affect the final outcome.

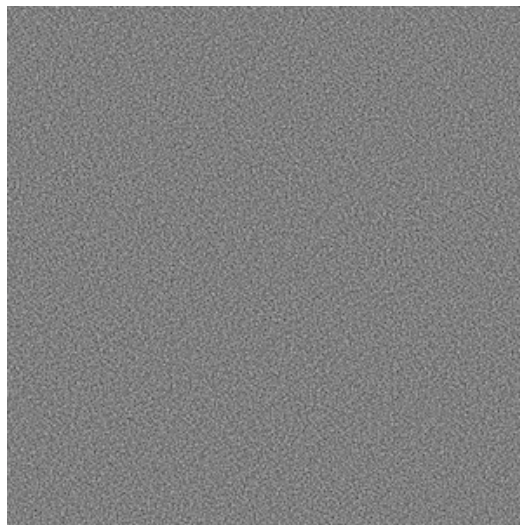


Figure 3.18: Regular Static. Image from <https://medium.com/@yvanscher/playing-with-perlin-noise-generating-realistic-archipelagos-b59f004d8401>.

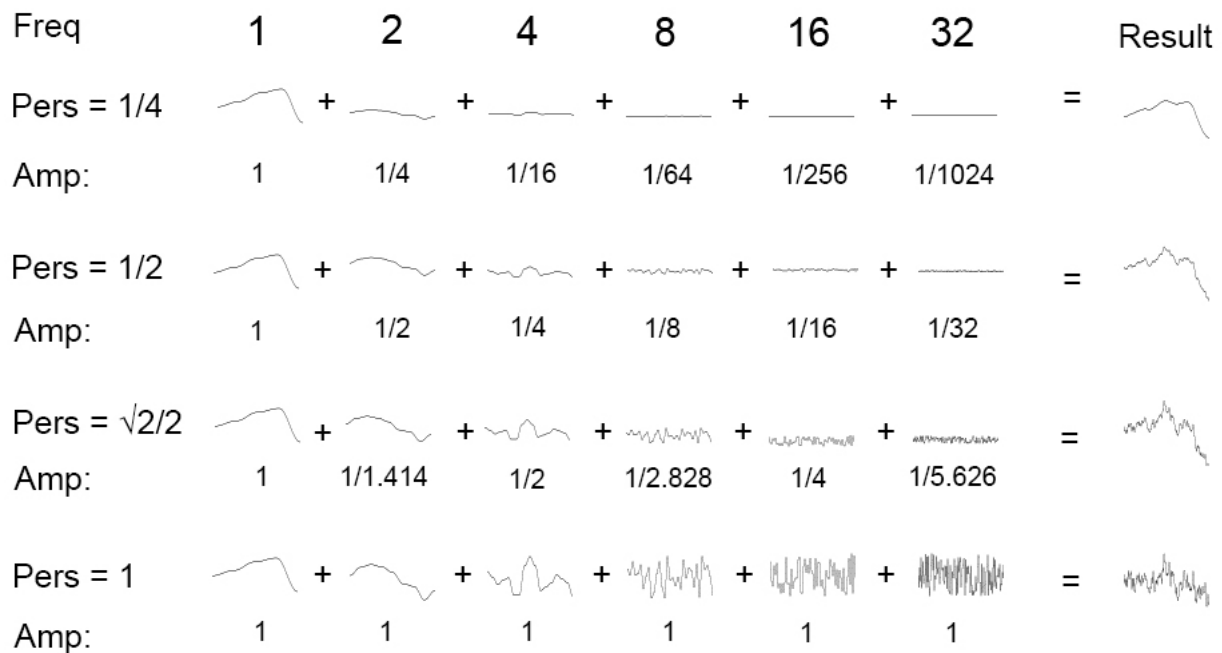


Figure 3.19: Adding different noise functions with different persistence values and the resulting Perlin Noise function. Caption - Freq: Frequency, Pers: Persistence, Amp: Amplitude

We already have a measurement that determines how much each octave contributes to the overall shape, Persistence, but we can also define a value that controls how much detail is added or removed at each octave, and by controlling detail, we mean adjust the wave frequency of each noise function being added. That measurement is called **Lacunarity**. A Lacunarity value of more than 1 means that each octave will increase the level of detail, because it increases the frequency. Lacunarity of 1 means that each octave will have the exact same level of detail and a Lacunarity of less than 1 means that each consecutive octave will get less detailed, with less frequency.

As was mentioned before, discrete noise functions need to be interpolated in order to smooth out the values and obtain outputs from non-integer *seeds*. A standard interpolation function takes three inputs: a and b which are the value to be interpolated between and x which takes a value between 0 and 1. The function returns a value between a and b based on the value of x . For $x = 0$, the output is a , for $x = 1$ the output is b .

There are several approaches to interpolate values, Linear Interpolation produces results with high performance rate, due to the simplicity of the method, but in exchange, the results itself are not the best looking, as we can see in Figure 3.20, especially when generating terrain, because it produces hard edges. It is however a good choice if used to generate landscapes in real time.

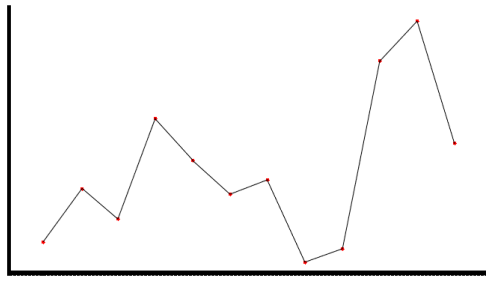


Figure 3.20: Linear Interpolation

Cosine Interpolation is a good compromise between performance and look, as it produces a very smooth interpolation between points, 3.21.

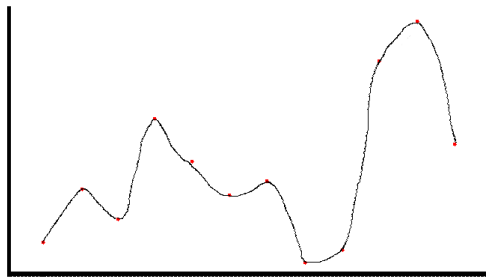


Figure 3.21: Cosine Interpolation

Until now we have been addressing the one-dimensional approach to the Perlin Noise algorithm concept, but it is however possible to implement the algorithm in any number of dimensions, although it becomes very computationally heavy the higher the dimension. To implement this algorithm in two-dimensions and use it to control terrain height, we first need to define a grid, where each grid intersection has a fixed random n -dimensional unit-length gradient vector. After that, we need to compute the dot product between the gradient vectors and their offsets. First we find the grid cell in which the candidate point lies, then we identify the 2^n corners and their associated gradient vectors. Next, an offset vector is calculated for each corner, which is the displacement vector from the candidate point to that corner. Finally we take the dot product between the gradient vector and the offset vector to the candidate point, which will be zero if the candidate point is exactly at the grid corner.

The final step, is, as we discussed previously, the interpolation between the 2^n dot products and it is performed using a function that has a first, and possibly second derivative zero at all the grid nodes. The output will approximate the dot product of the gradient of the node vector and the offset vector to the node, at points close to grid nodes, which will make the noise function pass through zero at every node.

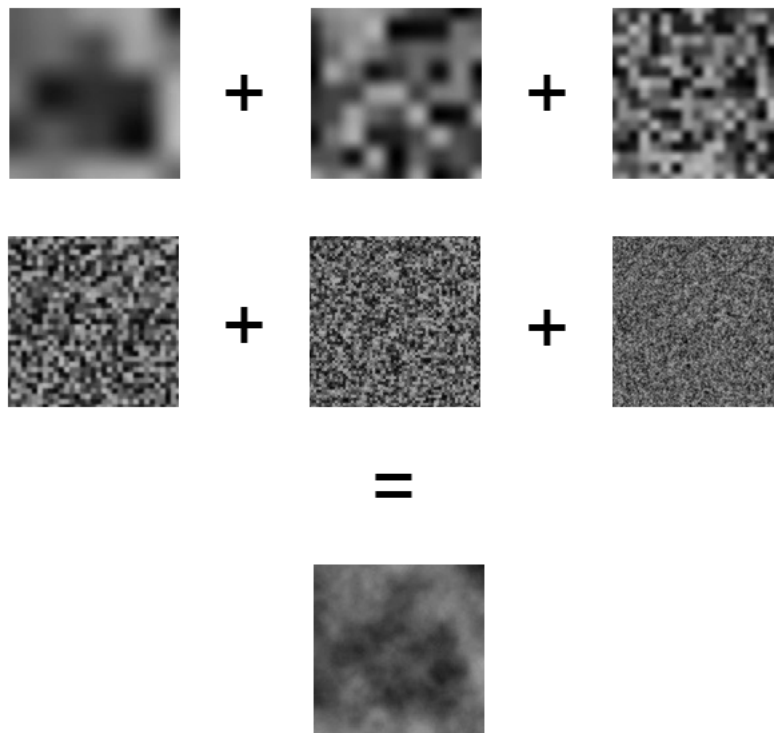


Figure 3.22: Same concept as Figure 3.17 but in 2D. Image adapted from <https://catlikecoding.com/unity/tutorials/simplex-noise/#a-patented>.

It is possible to achieve extremely good results generating terrain using Perlin Noise, with well defined parameters for each octave. The algorithm has complexity $O(2^n)$ for n dimensions, meaning it will require computation of 4 offset vectors and dot products in a two-dimensional grid and the double for the three-dimensional case. This obviously results in a computational overhead that needs to be addressed, specially when using noise higher than 2 dimensions, which we will use to give more features to our terrain, like caves and overhangs.

3.3.2 Simplex Noise

Limitations come from the use of a hypercube grid to divide the space. The appearance of square patterns can be noticed in the output. Also, triple interpolation make the cube's center more diffuse than the faces. Another problem that may rise, is the difficulty of computing analytical derivatives, as they get more expensive the higher the dimension.

To address the problems Perlin Noise had, Ken Perlin developed the Simplex Noise algorithm in 2001 [34]. The advantages are:

- Easier to scale to higher dimensions, has it achieves similar outputs to Perlin Noise with lower computational complexity, because it performs less multiplications and thus having an algorithm complexity of $O(n^2)$, for n dimensions.

- Has a continuous gradient which is well defined.
- No noticeable directional artifacts, which means it is visually isotropic (it has uniformity in all directions).

The main difference between Simplex Noise and classic noise, like the algorithm we previously explained, is the fact that classic noise interpolates between gradients at the surrounding hypergrid end points as opposed to Simplex Noise which divides the space into regular chunks called simplices⁴ and does not interpolate the corners at all. This is the main reason why the computational overhead for Simplex Noise is significantly lower, if we are in n dimensions and we use an hypercube, it will have 2^n corners, if we use a simplex, it will have $n + 1$ corners, so it reduces the number of data points. Thus, a simplex is the smallest possible space to grid cells. For the 1-dimensional case, the simplex is a line, so no difference here compared to classic noise. For the 2-dimensional case, simplex uses a triangle instead of a square. In 3D the simplex is a tetrahedron (3.23), instead of a cube.

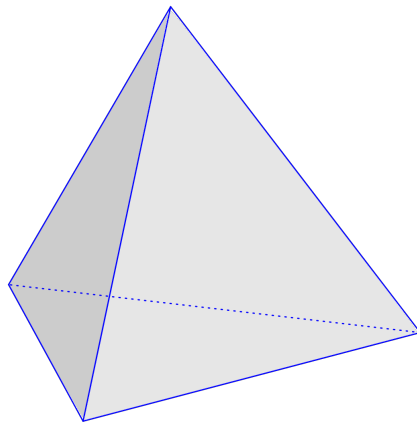


Figure 3.23: Tetrahedron. Image from https://commons.wikimedia.org/wiki/File:Euclid_Tetrahedron_4.svg

Following the line of thought we have been establishing until now, the final interpolation step is missing. Interestingly, we do not need to interpolate between simplex corners. Instead, we use a summation of contributions of each corner based on their distance from the sample point. The contribution is a multiplication of the extrapolation of the gradient ramp and a fall-off function⁵, that outputs zero when it reaches the opposite side of the triangle. This way, we

⁴A simplex is a essentially the name given to the most compact and simple geometric shape that if repeated, can fill an entire n -dimensional space

⁵A fall-off function is the name given to certain functions, in the context of noise development. These functions are used instead of regular interpolation because they are computationally cheaper to manipulate and their name comes from the fact that they are radially symmetric attenuation functions, in order words, they produce smooth slopes when reaching extremes.

make sure the influence of each corner only affects that corner's simplex, as it reaches zero when reaches the boundary. We use radial fall-off in the 2D case and spherical fall-off in 3D.

A fall-off function could be used, in theory, with a hypercube grid as well. The problem is the radial fall-off does not produce as good looking results as normal interpolation when dealing with squares, as shown on Figure 3.24.

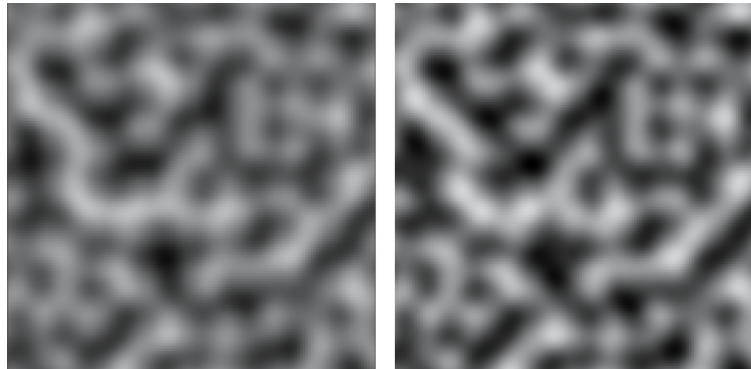


Figure 3.24: The left shows regular Perlin Noise and the right shows Perlin Noise that used a radial fall-off function. It is possible to notice the hard edges the right produces as opposed to the smooth gradient interpolation produces.

The biggest advantage of the use of a fall-off function instead of an interpolation function is that the contribution of each point is now independent, as they are just added to get the final value, which largely simplifies the calculation of derivatives. Figure 3.25 shows a representation of 2D Simplex Noise.

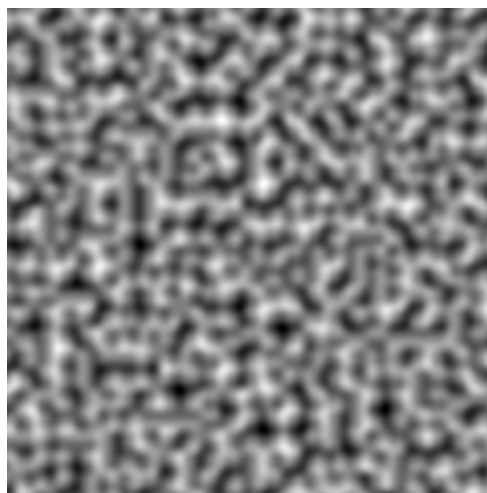


Figure 3.25: Simplex Noise. Image from <https://catlikecoding.com/unity/tutorials/simplex-noise/#a-patented>

3.3.2.1 Noise Usage in Procedural Generation

The use of noise to generate features may not be completely straightforward to the reader. In Figure 3.25, the representation is in 2D but it can be used to understand how a terrain using this noise would look like if it was represented in 3D. As it can be seen, the image is a gray scale ranging from white to black. The idea is to consider that these colours represent height, so white is the lowest and black the highest height value. With this in mind, we can start to understand why noise functions like this can represent terrain so well, because all the transitions between pure white and pure black are never abrupt, having a gradient between them and thus, a smooth variation in height values. By changing the frequency and amplitude of this signal, the density and height, respectively, we can adjust hills and general topography to our needs, depending on what type of terrain we intend to generate. When generating a Hot Desert, we know the topography is completely different from a Alpine Tundra, this is why different values for these parameters are needed.

Vegetation also uses noise to be placed randomly in our landscape with the same idea of placing hills or valleys. Considering Figure 3.25, vegetation would be placed only in spots where the height (or the colour black, in the image context) is greater than a certain predefined value, depending on what we want to place. For example, in a Savannah biome, which has a really low amount of trees, they would only be placed in locations that are represented black in Figure 3.25 and in a temperate forest, there is a higher amount of trees, so the placement is less strict, this time placing trees in black and grayer positions.

Caves and overhangs follow the same theoretical logic we just mentioned, related to vegetation and the terrain itself, but although these features could be generated using a 2D noise output, caves and overhangs, if they are generated also using noise, need a 3D output. This occurs because caves are carved inside the terrain, and the way to do so follows the same idea that was explained on the beginning of this subsection: we interpret the gray scale in Figure 3.25 as different height values along a plane, to give our terrain different elevations, and the places where caves are generated below the ground, are the places along the *z-axis* where the associated noise value (or colour in a gray scale) is related to a low height. This concept might be difficult to imagine, and impossible to represent in a picture, like the 2D version, but acts in the same way: white means a low height, black means a tall height, both along the (x, y) -axis, and now we apply this notion to the third axis.

Since the intention of this framework is to generate unique landscapes, one can argue that giving the framework the same exact sketch, it would always produce the same exact terrain. This is indeed true if the specified *seed* value is the same for both generations. A **seed** value, in this framework context, is a user predefined integer value used to initialize a pseudo-random number generator. A change in *seed* will result in an offset change of the output noise, as can be seen in Figure 3.26.

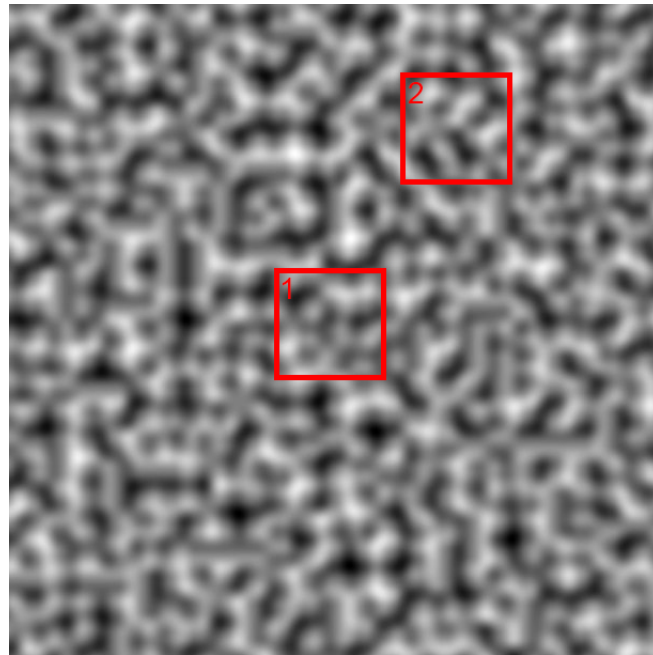


Figure 3.26: The red square marked with number 1 represents our terrain using the center location as the noise sample it will use. Changing the seed to other arbitrary integer will result in a change in offset from where the noise is sampled, marked with number 2.

As we can see, changing the seed will not alter the noise sample itself but rather the location from where the noise is sampled, considering the whole noise function output. This way we can obtain completely unique terrains between different users, because they choose a base *seed* on top of the sketch they desire for the terrain.

3.4 Clouds

Clouds, even though they are not related to a terrain or its generation, are a feature which is always welcomed when dealing with landscapes, as they add another visual feature to the final look of the terrain that can make a difference, depending on the way we want to experience this terrain. If the terrain is intended to be always used with a top-down orthographic view, including clouds would not be very useful since they would be obstructing the view. In the situation where the user experiencing the terrain is doing so from the perspective of someone who actually is inside the generated landscape, clouds can have a big impact on the experience.

Another important aspect about the clouds is we do not want them to cause a big computational overhead, since the terrain generation itself is already expensive on that regard. So, the idea, is to have a *png* image with the clouds, as we can see in Figure 3.27 and we want to project this image on a certain layer, above the terrain. We also want the clouds to slowly move, as if they were moving with the help of a breeze. It is very important that the player does not realize the clouds are just a static image, and that they are virtually always the same as it would just ruin

the immersion the clouds can provide. To do this, the idea is to have the position of the clouds, and the movement they have, related to the position of the player on top of our terrain.

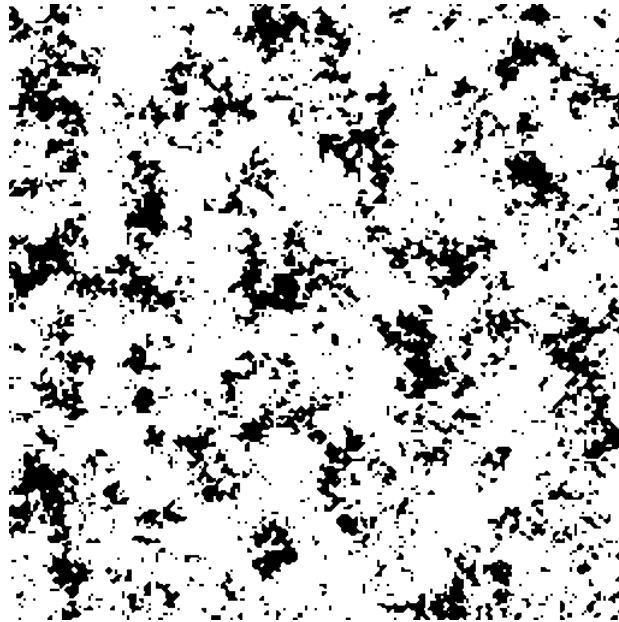


Figure 3.27: This is the *png* image used for the clouds texture. The clouds are obviously white but were painted black here, so we can visualize them and distinguish it from the background. Image from <https://minecraft.fandom.com/wiki/Cloud>.

3.5 Terrain Visualization

Although generating the terrain is the most important aspect of this thesis, because it is the final goal and the whole point of the work, it is of the most importance to provide the user, the one generating the terrain, some ways of experiencing and visualising the landscape he just created. We do not know what is the final goal of this terrain, what this stretch of land going to be used in. For this reason we find it mandatory to provide the user more than one way to look at the terrain, so he can look at the terrain the same way he would look if it was being used with the goal he desires.

The views we are including on the framework are: a top-down orthographic view, a free camera mode, where the user can fly and go to any place and a player perspective camera. The player perspective is a first person view inside the terrain, where the user has controls to walk around and jump to visualize all the details of the environment as if he was playing inside it. Also, it is in this mode the clouds appear and move according to the player movement, to give an extra layer of immersion to the one viewing the terrain.

3.6 Conclusion

After reading this chapter we hope the reader has now a solid understanding of how and what is trying to be accomplished and what was the thought process behind each decision. We reviewed every component that integrates the framework, and the ideas used to achieve it.

During the framework development, some decisions were made that changed the thesis' way, and we will use this conclusion section to expose these changes. Originally, the idea was to generate a terrain that was composed of a mesh created from a height map to shape the terrain's topography and also a colour map to give a texture to the mesh. This approach resulted in simple terrains with low level of detail and originated some problems and difficulties on the implementation regarding the division of the mesh in the various, user decided, biomes. Also, this implementation was being done using classic Perlin Noise, which is already implemented in the Unity API as a function in the *Mathf* struct⁶, and therefore the problems we mentioned in Section 3.3, inherently related to this form of noise, like the computational overhead associated, or the inability to have a 3D noise output to generate different structures, like caves, inside our terrain, were noted. To solve these issues, the approach was changed and instead, the framework was developed using voxels and blocks. This decision was made because voxel worlds have a higher degree of control over the elements that compose them, as opposed to generate and edit a continuous mesh and have it coherent in the end. Voxel terrains also open more possibilities in case the terrains are used in an application where the user will need to edit the landscape in real time, as it is simple to change the framework to accommodate these features. Changing the noise algorithm to Simplex Noise was also a major upgrade, since the boost in performance was significant and we were now able to use our noise output to generate different structures.

Following up, the next chapter will take a more in-depth look at how the ideas shown in this chapter were implemented in the Unity Engine.

⁶More information in <https://docs.unity3d.com/ScriptReference/Mathf.html>

Chapter 4

Framework Implementation

This chapter will explain the steps taken while developing the framework. The explanation will be made following the logic sequence of events that occur, in order to present a final built terrain. All the code was written using Unity Engine's scripting language, C#, and respective API. We will first make an overview of Unity and its basic features, for better understanding of the following explanations. The sequence of events we are showing also focuses more on the terrain generation phase, and features like clouds and the different ways of visualizing the terrain will be detailed after.

The sequence of events for generating a terrain is the following:

1. User sketching phase
2. Interpreting the user sketch
3. Based on the sketch size, determine the terrain layout, i.e. the number of chunks that will be generated
4. Depending on the positions determined by the sketch, generate the landmass
5. Carve cave systems, overhangs, terrain features
6. Generate vegetation

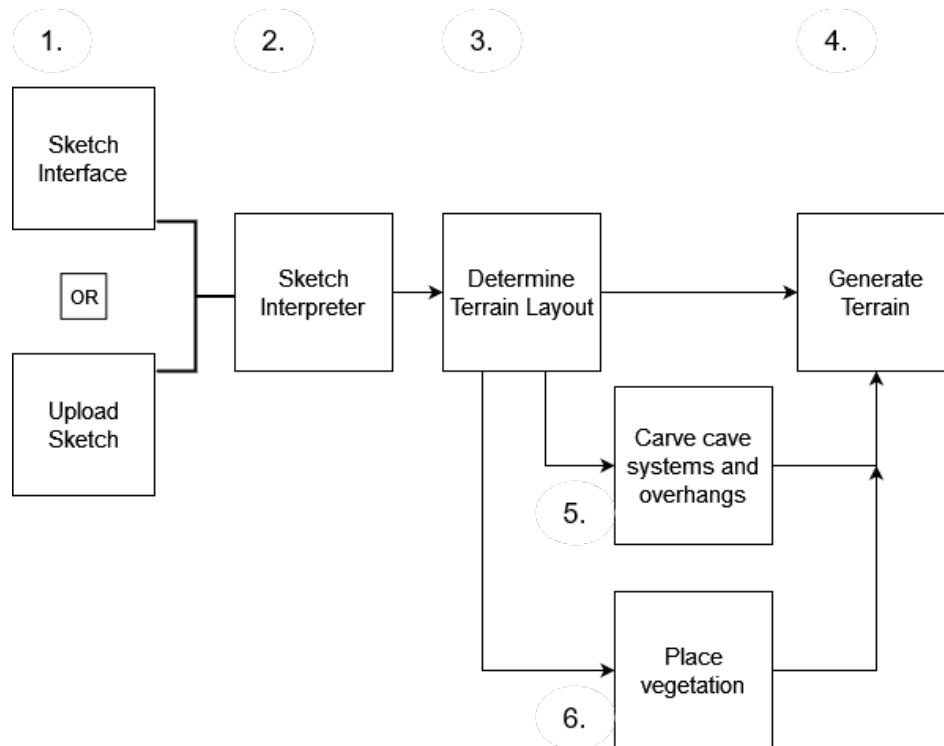


Figure 4.1: This image shows the sequence of events since the sketch phase until the terrain is fully generated. Each section will have another diagram with a higher level of detail

The next following sections will explain this sequence of events in detail.

The main controller scripts created to manage the generation of terrain in this framework, which will be used to explain how they all interact are the following:

- **Terrain Manager** - Main manager of the framework
- **Read Image** - Retrieve information from the user sketch.
- **Chunk** - Used to control chunks, has a Dictionary to store them, using their position as the key. Used for storing contents and create a mesh for rendering and collisions
- **Block** - Class containing information about one block. More sub classes of this class are created in order to add more different types of blocks to the framework
- **Mesh Data** - Store mesh data
- **Terrain Generation** - Class with generation methods of the terrain's topography
- **World Position** - Struct created to keep track of terrain positions
- **Noise** - Class which generates noise values

4.1 Unity Overview

Developed by *Unity Technologies*, Unity is a popular graphical engine used in the video game industry, but due to its flexibility and versatility it is also widely used in other areas, such as architecture, cinema, medicine, among others. This overview is important to explain key features and the engine's basic notions, in order to develop this framework. Only the most important features will be explained, as they serve as core concepts to the framework.

To start, Unity's objects placed in a scene are called *GameObjects*. They can have a wide variety of uses, not only being the objects we physically see in the scene, but they can be a light source or particle effect, or can just be a manager or controller for a mechanic or feature we may want to implement. *GameObjects* can contain scripts attached to them, to control actions on that *GameObject*. All *GameObjects* can have a wide range of components attached to them, depending on the use, but every object has an associated *Transform* component that is used to control the object's position, rotation and scale in the world scene.

Another important Unity feature is the possibility of using the built-in Editor to control public parameters and variables of the *GameObjects* we are using, without having to change the attached script. These parameters can even be changed at runtime, making the built-in Editor a powerful tool to define and test specific parameters in a very efficient way, without the delay of having to constantly compile and build the small changes in the code. This Editor can even be completely reworked via script so we can make the changes in the most optimized way.

It is also important to give a notion on how the Unity *Prefab* system works. It essentially allows us to create, configure and store any *GameObject* complete with all the components that are being used as well as property values and child *GameObject*, as a common *Asset*.

Finally, regarding the actual coding, it is convenient to mention the *Start* and *Update* loop, the most basic control structures a Unity script has. The first one is straightforward: it is a script function included (or not) in every Unity script that inherits from *MonoBehaviour*, which is a Unity prebuilt class that provides access to important control functions, that starts the execution of its contents in the very first frame after the start of the execution. The second, the *Update* loop, also inherits from *MonoBehaviour* and this function's contents run once every frame, and therefore is used to keep track of the project's logic, so it is totally dependent of the frame rate we are working on, but since generation times will not be a key factor for this framework, because we are mostly demonstrating this work and not trying to achieve peak performance, it will not be an issue. The *Update* loop will be used for the rendering of the final texture mesh and collision mesh the landscape will have.

4.2 User Sketching Phase

A simple UI was built for the user to produce a sketch with the characteristics it needs to be correctly interpreted. The main characteristics are size, colours used and format (*png*), as well as some configurations the image needs, to correctly display it with no compression, point filter mode and clamp texture wrap mode. The user starts by choosing a predefined size for the final terrain, having the possibility to choose between six different predetermined terrain sizes, $[32 * 32]$, $[64 * 64]$, $[128 * 128]$, $[256 * 256]$, $[512 * 512]$, $[1024 * 1024]$, having in mind that 1 pixel in the canvas corresponds to the position of a block on the terrain. After choosing terrain size, the user paints a white canvas, using the available colours. At this stage the user also needs to choose an integer to be the terrain *seed*. When the user is satisfied with his sketch, he saves the image.

The canvas is basically a blank plane object, and the brushes are *Prefabs* created with the correct colours already assigned. In order to give the impression of actually be painting, we use Raycasting, to cast a ray until it reaches the initial plane object's collider, and when it does, it instantiates an object for every frame the user is inputting the *Mouse1* key. In the end, all the objects on top of the plane are captured by a render camera that saves it on a *Render Texture*¹ object and this texture is then converted to a *png* image file, to later be used. Figure 4.2 shows a preview of what the UI looks like while being used, in Unity's Game View.

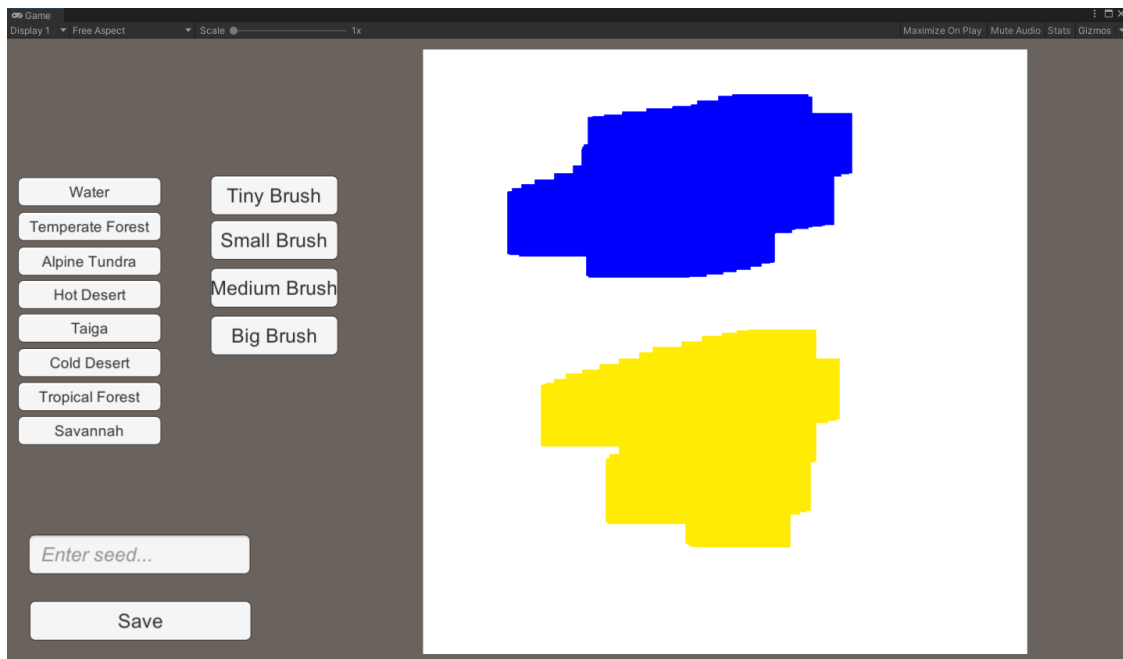


Figure 4.2: UI to make sketches while being used: A button for each biome colour, four different brush sizes to make sure we can sketch with necessary detail, an input field for the *seed* and a save button when the sketch is complete

¹A *Render Texture* is a type of *Texture* that Unity can create and it is updated at run time. More information in: <https://docs.unity3d.com/Manual/class-RenderTexture.html>

Another option the user has is to place a previously made sketch into the Unity project folder, as long as this image possesses the already mentioned correct characteristics: the sketch is a square where the side needs to be a power of 2, greater than 32 (in the sketch UI we only have the possibility to choose between six different sizes because these are the sizes the framework is prepared to generate out-of-the-box, however, the framework can, in theory, admit any sketch size, as long as it is a power of 2, and with an easy manipulation to the function where chunks are generated, will produce a terrain with that size), painted with the exact intended colours which are associated with each biome. Also, when importing the image sketch into the project, one needs to be sure the image has no compression. If the image is compressed, it will generate artifacts and blurs between each colour (if the sketch has more than one colour) and that results in a range of colours that are not recognized by our script, what happens will later be explained. The *seed* will be fed into the system in the next phase.

The next list will present the colours that can be used on the sketch and their respective RGBA² and HEX³ values:

- Blue - **RGBA**: (0, 0, 255, 255) - **HEX**: #0000FF
- Yellow - **RGBA**: (255, 255, 0, 255) - **HEX**: #FFFF00
- White - **RGBA**: (255, 255, 255, 255) - **HEX**: #FFFFFF
- Green - **RGBA**: (0, 255, 0, 255) - **HEX**: #00FF00
- Black - **RGBA**: (0, 0, 0, 255) - **HEX**: #000000
- Cyan - **RGBA**: (0, 255, 255, 255) - **HEX**: #00FFFF
- Magenta - **RGBA**: (255, 0, 255, 255) - **HEX**: #FF00FF
- Red - **RGBA**: (255, 0, 0, 255) - **HEX**: #FF0000

4.3 Interpreting the User Sketch

As said before, the user sketch consists in a $[w * h]$ colored *png* image. Unity has a structure integrated in its API which is named *Color*⁴, used to represent colour variables. It has built-in predefined colours, for a fast use. The colours used for the sketch are exactly the same, in terms of its RGBA colour values, as the ones the API uses. So for example, Unity's *Color.blue* has the RGBA value of (0, 0, 1, 1) and *Color.red* has the RGBA value of (1, 0, 0, 1). It is important to add that Unity's *Color* type uses a scale from 0 to 1 as opposed to 255, meaning it has float

²RGB define a color in RGB color space (Red, Green, Blue) and A, the alpha component, defines transparency.

³The HEX value represents the exact same as the RGBA but each different colour has an Hexadecimal number associated with said RGBA value

⁴More information in: <https://docs.unity3d.com/ScriptReference/Color.html>

values to describe each RGBA component byte value. Unity also has the *Color32* type, to use colours in the 32 bit format, as was shown in the previous section. The only colour we used that was not part of the Unity API is yellow, since Unity's yellow is a specific colour (RGBA is (1, 0.92, 0.016, 1)) and we wanted the simplest colours possible. The ones used for the sketch have the same RGBA values as the Unity's API static colours because it made the process of interpreting them easier, as the colours in itself are not important and act only as a simple way to keep track of the positions the user wants for each biome.

The framework starts the routine in the *Start()* function present in the **TerrainManager** script which is attached to the *TerrainManager*, 4.3 *GameObject* placed in the scene. This function essentially calls the **ReadImage** script.

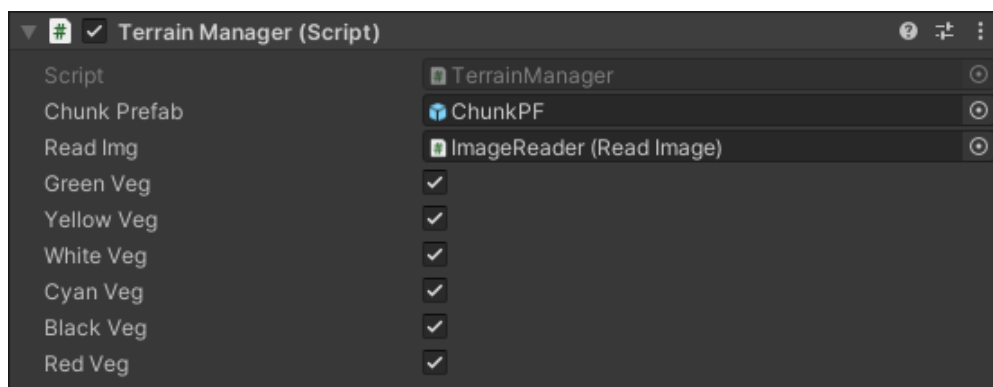


Figure 4.3: *Terrain Manager GameObject*, with the **TerrainManager** script attached, controlling the execution flow and a Prefab that will be used for each chunk. We can also see some *bools* that control vegetation generation, they will be later explained

A *GameObject* called *Image Reader*, Inspector view shown in Figure 4.4, which has the **ReadImage** script attached, placed in the scene, controls, as the name implies, the reading of the sketch previously drawn by the user. The **ReadImage** script starts by creating a colour map from the sketch, using each pixel's colour. The width and height of this image, chosen in the previous phase, is saved. In this explanation example, we are choosing a terrain with the size [256 * 256]. Then, the position of each pixel of the image is verified and depending on the colour, each position is saved in a public list dedicated to that colour. If, as was mentioned before, something occurs that results in a sketch with colours which are not accounted for, these positions will not be saved to any list. To prevent the framework from getting its execution stuck because it has no way of dealing with unrecognized colours, when the terrain starts to be generated, these positions will not be affected by any kind of terrain modelling algorithm, and will instead have instantiated a column of stone blocks, with the maximum height any block can have on the terrain. Having a list of positions for each colour, in the end, the **TerrainManager** script is called to initiate the process of defining the terrain layout.

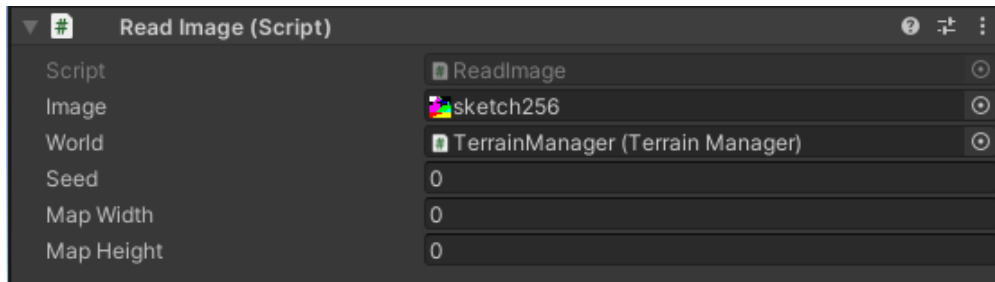


Figure 4.4: *Image Reader GameObject*, with the **ReadImage** script attached, taking a test sketch from Figure 3.2 and the **TerrainManager** script, also keeping track of the width and height of the image, as well as the user decided seed

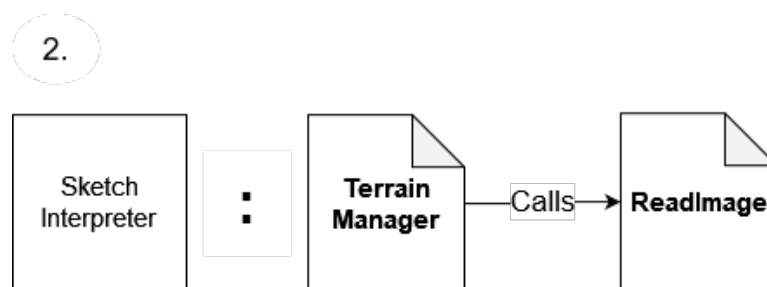


Figure 4.5: This diagram shows the logic behind the sketch interpreter as well as the scripts that are being used to do it.

4.4 Determine the Terrain Layout

Following the sequence, in this next phase, the **TerrainManager** script, after it called the **ReadImage** script to interpret the sketch, continues its execution to create the correct number of chunks with the correct size, based on the **ReadImage** script output. As said previously, we are using a fixed size of 16 blocks for the side of each chunk.

The terrain layout is determined, first checking if the *width* and *height* of the terrain is a power of 2 and then checking what the size actually is, so it can call the function that generates the chunks properly. Although, if anyone using the framework needs a terrain for any other size (that is a power of 2 greater than 32), it is in this phase that the code should contemplate the change. Figure 4.6 describes how the chunks are instantiated, it shows a scheme of the terrain scene from above, one with the size $[32 \times 32]$ and the other $[64 \times 64]$. Figure 4.7 shows the $[32 \times 32]$ version, with one chunk highlighted, inside the Unity Scene View.

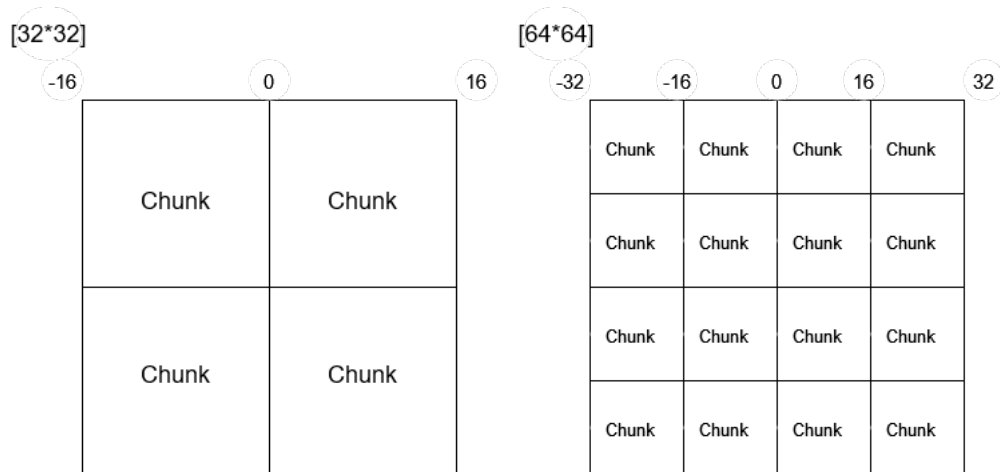


Figure 4.6: Scheme of a terrain to show how the chunks are layed out. It shows the 32 and 64 side size versions, but the logic is the same for any size above.

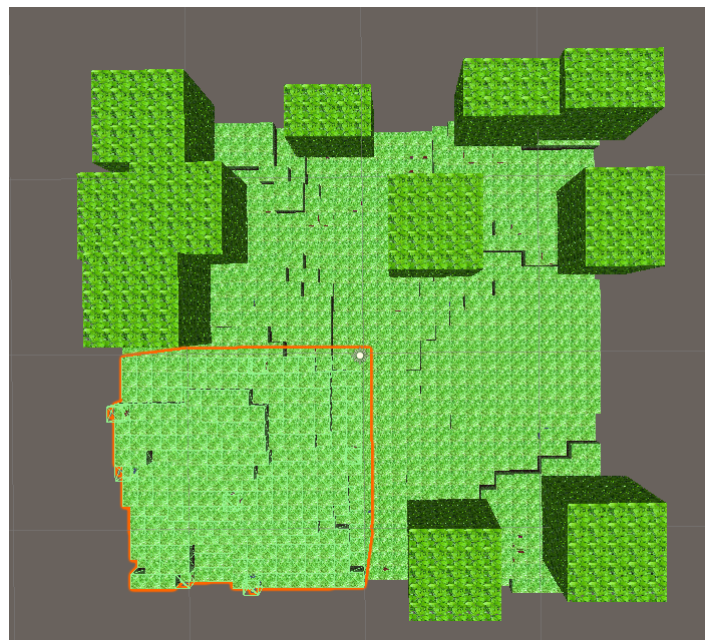


Figure 4.7: Here we can see the same scheme as in the previous figure, this time using an actual piece of terrain with one of the chunks highlighted.

Then, the chunk objects are created and added to a chunk dictionary⁵ using its world position as the key. The *Prefab* that is instantiated for each chunk is still empty but has the components that will be used to generate textures and collision mesh, as we can see in its Inspector view, in Figure 4.8, Mesh Filter⁶, Mesh Renderer⁷ and Mesh Collider⁸. The Mesh Filter has the goal

⁵For more information on C# Dictionaries, visit <https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2?view=net-5.0>

⁶<https://docs.unity3d.com/Manual/class-MeshFilter.html>

⁷<https://docs.unity3d.com/Manual/class-MeshRenderer.html>

⁸<https://docs.unity3d.com/Manual/class-MeshCollider.html>

of taking a mesh, in this case, the *tiles* asset, and passing it to the Mesh Renderer, so it can take the geometry and render the mesh at the specified *Transform* position. The *tiles* asset is not passed as it is, we had to first create a Unity Material from it. Materials⁹ in Unity are closely linked to shaders, as they are always used in conjunction, and they both describe the appearance of surfaces. It is created as a transparent cutout diffuse shader (This shader was used because it was ideal: it cannot have partially transparent areas and objects using it cast and receive shadows. Diffuse computes a simple, Lambertian¹⁰, lighting model) and it uses the *tiles* map as the base. The Mesh Collider, as the name implies, takes a mesh and builds a collider with it, so our terrain can be solid and interactable with other solid objects. A texture and collision meshes are added in the next phase, after the **TerrainGeneration** script is called.

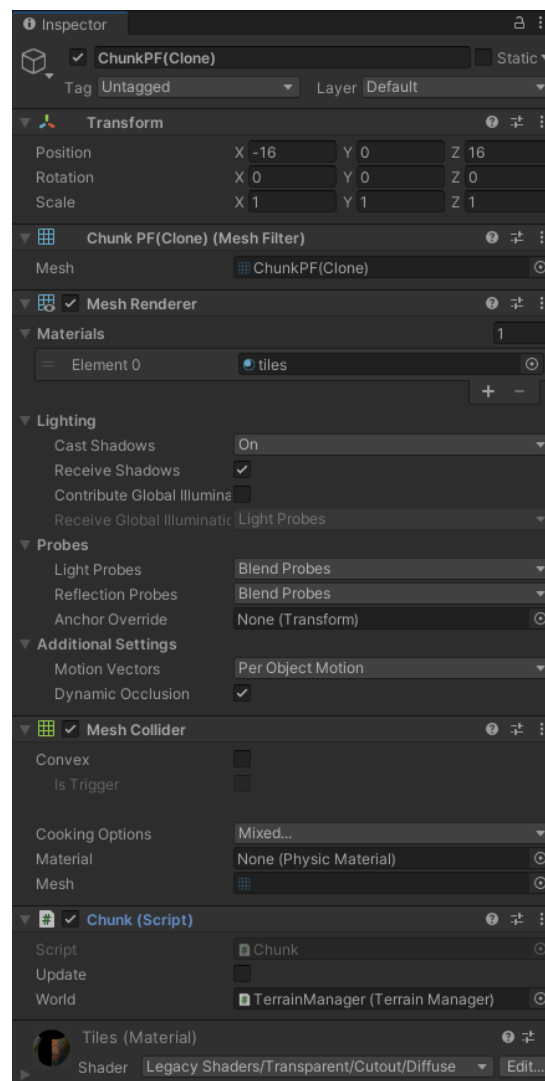


Figure 4.8: *Chunk Prefab*, with the **Chunk** script attached, a Mesh Filter, Renderer and Collider components as well as the tile map that will be used for textures.

⁹<https://docs.unity3d.com/Manual/materials-introduction.html>

¹⁰Lighting on these surfaces decreases as the angle between it and the light decreases, it does not rely on the camera's position or movement

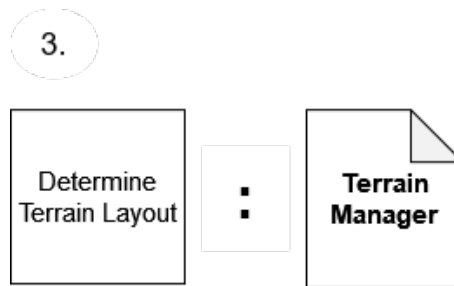


Figure 4.9: As we can see the terrain layout phase is determined using the Terrain Manager script

4.5 Generate Terrain

The chunks now need to be filled with blocks, with its respective textures and collisions. The **Terrain Manager** script calls the **Terrain Generation**. This script is responsible for the organization of how the blocks should be placed inside each chunk. It starts by looping all the (x, y) positions inside each chunk, in order to generate a column of blocks for each position in the terrain. The number of blocks and which blocks need to be placed in each column depends on which biome the (x, y) position is located, so we compare it with the lists of positions we created when the user sketch was read, in order to call the respective function that creates that specific biome.

Before setting the blocks, some base values need to be setup for each block type we are including in our terrain. In "normal conditions"¹¹ the **Stone Block** is the base terrain block, has it is the only block present across all biomes. The terrain topography is carved using this block as a base and all other blocks are set on top of Stone. To start, in each biome, Stone needs a value for its base height (*stoneBaseHeight*), the lowest layer a Stone block can be in. Next, a base noise is defined (*stoneBaseNoise*), in other words, a value that controls the scale of the noise to add to the base height. Finally a value for the height of the noise (*stoneBaseNoiseHeight*), for example if this value is set to 5, the maximum difference between a peak and a valley is 5 blocks. The next important values to setup, are also related with Stone blocks, but a distinction needs to be made. These are Stone values related with the mountains that occur in each biome, so they will be more or less significant depending on the biome we are setting blocks in, as opposed to the previous ones, that are meant to control base values. So, a mountain height value is setup (*stoneMountainHeight*), as well as a value to control the frequency of mountains (*stoneMountainFrequency*) and a minimum height value a stone block that belongs to a mountain can have in this biome (*stoneMinHeight*). Table 4.1 shows what values for the base Stone and for the mountains were chosen for each biome.

¹¹This expression is between quotation marks because "normal" is considered in this work's context a diverse terrain, that includes all types of biomes available. If the user decides to generate a terrain which is a Water biome, for example, in its integrity, the main block used is the Water block, on top of a stone layer.

Table 4.1: Mountain values marked with "-" mean that they are irrelevant, because mountains are not generated on those biomes. A pattern can be observed regarding the *stoneBaseNoise* because it presented the best results in biomes that did not have mountains, using different values for *stoneBaseHeight* to make the adequate changes between terrains. Only two biomes can be considered mountainous has the *stoneMountainFrequency* can tell, the Alpine Tundra and Taiga, with mountain heights reaching the tallest values.

	<i>BaseHeight</i>	<i>BaseNoise</i>	<i>MountainHeight</i>	<i>MountainFrequency</i>	<i>MinHeight</i>
Water Masses	-30	0	-	-	-15
Hot Desert	-28	0.05	-	-	-12
Cold Desert	-28	0.05	-	-	-12
Taiga	-24	0.05	48	0.035	-8
Alpine Tundra	-24	0.05	48	0.039	-8
Temperate Forest	-28	0.05	-	-	-12
Tropical Forest	-28	0.05	-	-	-12
Savannah	-28	0.05	-	-	-12

After setting these values, we use a variable called *stoneHeight* which has the value of the largest integer smaller or equal to the *stoneBaseHeight*, making use of Unity's math library *FloorToInt*¹² function. Now it is time to use our implemented noise function, as discussed on the previous chapter, in order to give our terrain a realistic look, with hills and mountains. We will call this function *GetNoise()*, and it will later be explained in greater detail how it is implemented. This function takes five different parameters to produce an output. The coordinates where we want to find the respective noise, *x*, *y*, *z*, a value for *scale* and a *max* value the function can output. To the *stoneHeight* value we just created, we sum the value which the *GetNoise()* function will output, on those coordinates, with the *scale* being the *stoneMountainFrequency* and the *max* being the *stoneMountainHeight* we just defined. The values control the height the terrain features will have. It is convenient to point out that the biomes which can be observed on Table 4.1 to not have specific mountain values for their variables, do in fact have a value on the code, but is in this context completely negligible, because these biomes do not have actual mountains but make use of those variables to make small hills, or in the desert biome case, dunes. These are features which add more detail to the environments.

Next, we make sure the stone height can never be less then the value we established previously for minimum height (*stoneMinHeight*) and we add the stone base noise to our variable. The stone base noise is obtained from the *GetNoise()* function we mentioned, this time using the *stoneBaseNoise* and *stoneBaseNoiseHeight* variables as *scale* and *max*, respectively. Finally, we need a value to control the height of blocks placed on top of the stone. It depends completely on the biome we are in but if it is a desert biome, for example, we want sand on top, so we

¹²For more information on the *FloorToInt* function, visit: <https://docs.unity3d.com/ScriptReference/Mathf.FloorToInt.html>

need the *sandHeight* variable. Exactly as the stone, we need the variables *sandBaseHeight*, *sandNoise* and *sandNoiseHeight* so we can produce a noise output to add to our *sandHeight* variable. The rest of this explanation will be given as if we were generating a desert, so we can have coherence.

Before finishing the terrain generation phase, we will detour in order to give insight in our *GetNoise()* function, which calls the user defined *seed* value and uses Unity's *System.Random*¹³ in order to generate a random value using the *seed* variable as a base. Next, the function determines, using this random value generated from the *seed*, an *offset* value for *x* and *z* axis. The generated value is in the range $[-100000, 100000]$, because while testing, it was noted that if the generated number was too big or too small, the results would not be satisfactory, some times not producing any change to the terrain. The *GetNoise()* function calls the **Noise** script which returns the noise value, but to each coordinate we add the correspondent *offset*, which will give us another sample of the noise, related to the user chosen *seed*. The next subsection will give insight on the **Noise** script.

4.5.1 Simplex Noise

A Simplex Noise function takes 3D coordinates to generate a noise value. The function that was just mentioned takes the coordinates to sample (which, as been said, we offset, using our *seed* value, to have unique results), as well as a scale value which is used to multiply the coordinates: a smaller value samples a smaller area originating in smooth noise suitable for mountains with long distances between the peaks and valleys but a larger value will make more frequent bumps and dips. The last parameter is the max of the return value, it will always output an integer between zero and this parameter.

Simplex Noise's implementation usually takes four distinct steps: coordinate skewing, simplicial subdivision, gradient selection and kernel summation. In order to better understand this algorithm, we will explain the case for each dimension.

4.5.1.1 1-Dimension Case

Given a coordinate, we want to start by finding its lattice point¹⁴, and since in this case the simplex and hypercube grid (the Perlin Noise variant) are equal, the lattice point to the left of the input is its integer portion.

As mentioned in 3.3.2, we need a fall-off function to control the contribution of the corner of each simplex. The distance to the point we want to sample is the contribution of each corner. Calculating the distance would be straightforward in this case, but in higher dimensions we

¹³For more information on the *Random* function, see <https://docs.unity3d.com/ScriptReference/Random.html>

¹⁴A lattice point is a point in a Cartesian coordinate system such that both its *x*- and *y*-coordinates are integers.

need to use square root operation which would result in a high computational overhead which we want to avoid, so the function needs to be adequate even in higher dimensions.

The simplest fall-off function is $1 - x^2$ with first and second derivatives $-2x$ and -2 respectively, and thus inadequate since for $x = 1$, they are not 0. If we square it, $(1 - x^2)^2$ also does not work, since the second derivative is equal to 8 when $x = 1$. Finally, $(1 - x^2)^3$ is our fall-off function because both derivatives drop to zero when $x = 1$.

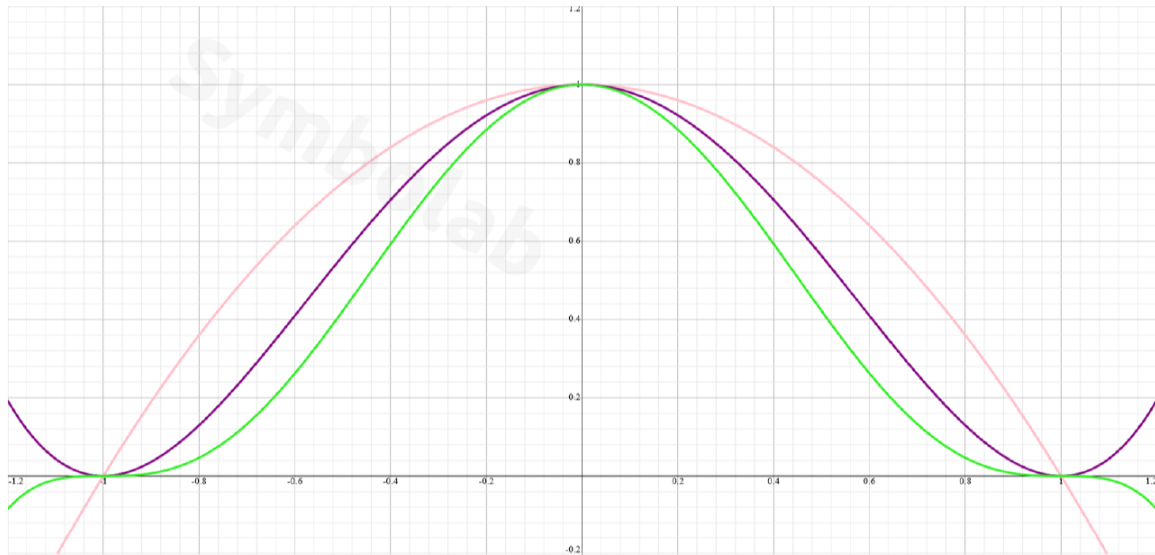


Figure 4.10: Mentioned fall-off functions graphs, green is $(1 - x^2)^3$, purple is $(1 - x^2)^2$ and pink is $1 - x^2$. Image made using <https://www.symbolab.com/>.

Before summing both final end results, we first need to hash¹⁵ both ends into a pseudo-random gradient direction, which, in the 1D case, can just take the positive or negative directions. The hash is implemented using a permutation table. The gradient value is computed by taking the dot product of the gradient vector and the vector from the corner to the point we are sampling, which for 1D, the vectors are actually just a value.

All that is left is to calculate the maximum value the fall-off function can reach, where the gradients on both ends are pointing towards each other. So the maximum is $2x(1 - x^2)^3$ when $x = 1/2$ which is $27/64$. If we multiply the final value (the sum of both final results, each one multiplied by the gradient vector) by $64/27$ we reach a noise value in the interval of $[-1, 1]$ (in other words, we normalized the interval). The following picture shows a representation of 1D simplex noise.

¹⁵Hashing is the process of using a function or algorithm to map object data to some representative integer value designed to solve the problem of needing to efficiently find or store an item in a collection.

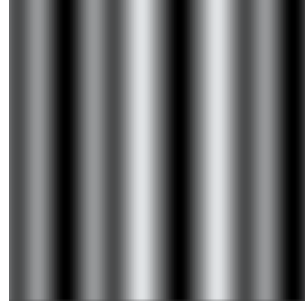


Figure 4.11: 1D Simplex Noise. Image from <https://catlikecoding.com/unity/tutorials/simplex-noise/#a-patented>

4.5.1.2 2-Dimension Case

For the 2D case, we again start by applying a fall-off function, but this time our coordinates include the second dimension y . For this reason, coordinate skewing no longer can be done using the method we used before. 2D space can be tiled with equilateral triangles or with squares and we want to convert between these two tilings by skewing and unskewing our coordinates, so we can be sure in which simplex we are in, for any point in space. A square divided along its diagonal produces two isosceles triangles, and by compressing those triangles along the same diagonal they become equilateral. This is why we skew the square grid and split the squares, to end up with an equilateral triangular grid. To scale along the diagonal, we need to find where each point of the triangle lies along that line. For every coordinate (x, y) this is equal to $x + y$. To downscale, we need to sum to each coordinate the skew factor, s which is equal to $(x + y) * F$, F being the scaling factor that converts triangles to squares, with the formula

$$F = \frac{\sqrt{n+1} - 1}{n} \quad (4.1)$$

n being the number of dimensions we are using. So $F = \frac{\sqrt{3}-1}{2} = 0.366025403$.

We are now able to find the lattice points of each input coordinate after we add the skew and this way find which simplex cell we are in. To get the actual corner points for determining their distance, we have to unskew them in a similar fashion, where the unskew u is equal to $(lx + ly) * G$ with (lx, ly) the lattice points we just calculated and G the factor that converts squares to triangles, with the formula

$$G = \frac{1 - 1/\sqrt{n+1}}{n} \quad (4.2)$$

If $n = 2$, $\frac{1-1/\sqrt{3}}{2} = 0.211324865$.

The next step is to determine which of the two triangles inside the cell we are in: if the fraction of our x coordinate is larger than or equal to the fraction of the y coordinate, we are in the bottom triangle, with coordinates $(0, 0)$, $(1, 0)$, $(1, 1)$. The top triangle has coordinates $(0, 0)$, $(0, 1)$, $(1, 1)$

So, naturally, these triangles in skewed coordinates would be $(-g, -g), (1-g, -g), (1-g, 1-g)$ and $(-g, -g), (-g, 1-g), (1-g, 1-g)$, g being our unskew factor 4.2 for $n = 2$.

We finally need to get the gradient vector and compute the dot product to find the contribution from the three corners, this time using a different gradient function, adequate for taking (x, y) coordinates. In this case, gradient can have 8 different directions, and so we convert the low 3 bits of hash values to obtain the sample value. All that is left is to find where the maximum value of the fall-off function is, just like in the 1D case, but this time in a triangle. The edge length is $\frac{\sqrt{6}}{3}$ and halfway along the edge we get $2x(\frac{1}{2} - x^2)^3 = \frac{\sqrt{6}}{81}$, with $x = \frac{\sqrt{6}}{6}$. The length from a corner to the center of the triangle is equal to its edge length multiplied by $\frac{\sqrt{3}}{3}$, so we have $3x(\frac{1}{2} - x^2)^3 = \frac{125\sqrt{2}}{5832}$, with $x = \frac{\sqrt{2}}{3}$ and this value is larger than the edge maximum, so in the center we achieve our theoretical maximum, as we can see in Figure 4.12. To end, we just need to multiply the final noise sample by the maximum's inverse and obtain a value in the range $[-1, 1]$.

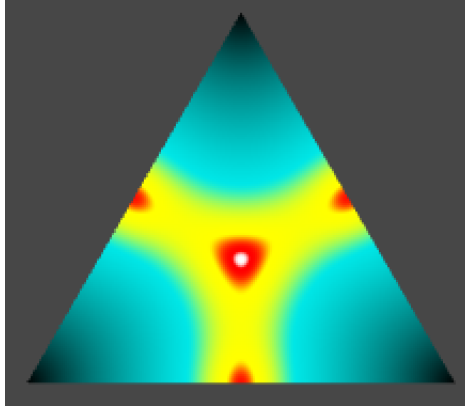


Figure 4.12: Maximum possible values for the fall-off function in a triangle. White is the highest. Image from <https://catlikecoding.com/unity/tutorials/simplex-noise/#a-patented>.

4.5.1.3 3-Dimension Case

This case basically works like the 2D case but we are dealing with a cube grid and our simplices are tetrahedra. However, as we cannot tile 3D space with a regular tetrahedron, we will work with skewed tetrahedra. Just as we divided a square into two triangles, we can split a cube into six tetrahedra, as shown in Figure 4.13. We can also note that all the simplices have a common edge (A,G), which is the cube's main diagonal. Following the same logic we did for the 2D case, we this time include a third z -axis, and to figure where, along the diagonal line each point lies for any point (x, y, z) , this is equal to $x + y + z$. The skew and unskew factors that make the conversion from tetrahedra to cubes and the reverse, using the Formula 4.1 we used before, this time with $n = 3$, we get $F = \frac{1}{3}$. Naturally, the unskew part, to be back in (x, y, z) space is also done in the same way, using Formula 4.2 we have $G = \frac{1}{6}$.

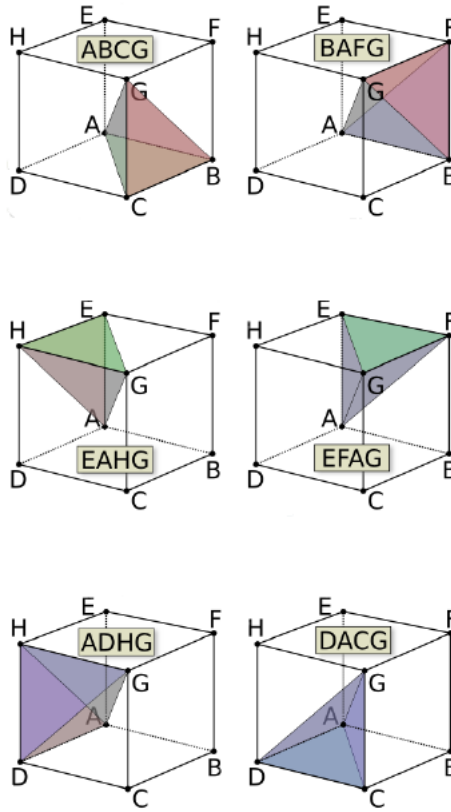


Figure 4.13: Division of a cube into six tetrahedra. Image adapted from https://www.researchgate.net/figure/Cubic-volume-division-into-6-tetrahedral-cells_fig1_269666018.

To determine in which simplex we are in, the process is exactly the same as in the 2D case but we need to factor in the new third component. We start by finding the closest corner on the main axes and then repeat the process and add the second closest corner, to get the fourth and last corner.

The gradient finding is equal to the previous case, this time converting low 4 bits of hash code into 12 simple gradient directions, but we need to find the maximum the fall-off function can be in a tetrahedron so we can get a sample in the $[-1, 1]$ interval. As the shortest edge length is $\frac{\sqrt{3}}{2}$, we achieve $2x(\frac{1}{2} - x^2)^3$ where $x = \frac{\sqrt{3}}{4}$ is $\frac{123\sqrt{3}}{8192}$ and thus our scale factor is $\frac{8192\sqrt{3}}{375}$. We should also take into consideration that the 3D gradient array has vectors of length $\sqrt{2}$, and so it is factored into the scaling factor, making it $\frac{8192\sqrt{3}}{2^2 \cdot 375}$. As in the previous example, we now obtain our noise value in the range $[-1, 1]$ by multiplying the noise sample with the inverse of the maximum we just found.

4.5.2 Column Generation

We now understand how the **Noise** script generates noise values and how the *GetNoise()* function handles this output to be used. Picking up where we left, we now have the *sandHeight*

and *stoneHeight* variables to control the blocks which will be placed on our terrain.

The height positions along the z – *axis* are cycled and we start by comparing said value to the *stoneHeight* variable we just defined to properly lay the correct foundation to our terrain. To clarify, for every Chunk, for all the (x, y) positions that were already cycled, for every height value z , if it is below the *stoneHeight*, we place a **Stone Block**. Next we do the same for the block that sits on top of Stone, as we discussed on the previous section, the sand variable, so we can place the **Sand Blocks** on top of the stone, and every position above the *sandHeight* variable will be filled with **Air Blocks**. Tables 4.3 to 4.10 present all the soil variables' values for each biome that can be used in the terrain.

The function that is called to set blocks takes the coordinate we want to set the block in, which block we want to set and the chunk we are in. It can also take a final *bool* parameter which is defaulted to *false*, which gives the option to make the block being placed replace other blocks already placed on the same position.

It starts by updating the coordinates position to the position inside the chunk and then checks if these positions are inside the intended chunk using a function from the **Chunk** script. This is a simple function that takes the coordinate and checks if its greater than the chunk size or less than 0, and if it is, means that the position is outside the chunk we are working in. Once we determine we are inside the intended chunk, we call a function inside the **Chunk** script to actually set the block. This function verifies if the coordinates are in range of the intended chunk again, and sets the new block if it is, but if it is not, it now calls a function in the **TerrainManager** script to handle the situation.

Since we do not know to which chunk the coordinates belong, this function basically takes the coordinates and calls a new function inside of the **Chunk** script which returns in what chunk those coordinates indeed are, by cycling through our chunk dictionary, comparing the given key (the coordinate) to the corresponding chunk. This time, as we most definitely know in which chunk we are, we again call the function which sets blocks, inside the **Chunk** script, that will finally place the intended block on the correct position. To finish, when all the blocks are placed, a flag called *update* is changed to *true*.

This *update* variable essentially controls Unity's *Update()* function inside the **Chunk** script, which was explained in the Engine's Overview, section 4.1. While it is *true*, it calls once every frame a function called *UpdateChunk()*. By this time, we essentially have all our terrain completely laid down, but we do not have a mesh for the visuals, or collisions, and this is what is generated by the update function every frame. The next subsection will detail how the mesh generation phase occurs.

4.5.3 Mesh Generation

The *UpdateChunk()* function starts by creating a mesh data variable of the type *MeshData*¹⁶. In Computer Graphics, a polygon mesh is a collection of vertices, edges and faces that defines the shape of a certain object. It essentially is the visual representation of what composes the terrain. We are also creating an additional collision mesh, just so we can have the possibility of using the generated terrains to include in any form of media which might need a walkable and interactable terrain, like a video game. Both meshes are separate but consist on the same polygons: for each voxel's face, we want to render a square, so we render two square triangles adjacent in the hypotenuse, making six squares (or quads), or twelve triangles.

UpdateChunk() compiles all the placed blocks in the *meshData* variable, to define what textures should be rendered, based on each block's data. As an example we will use Figure 4.14. As we can see, the block marked with red only has 3 faces that can be seen by the user, because the other 3 are adjacent to other blocks and thus hidden. Since they are hidden, there is no need for the engine to render these faces, it would just create a huge computational overhead with no benefit. Figure 4.15 exemplifies exactly how much of an unnecessary burden rendering useless meshes can create. This is what each block's *Blockdata* consist: a function inside the **Block** script that determines which are the faces that are adjacent to other solid blocks in order to only render a mesh data for visible block faces. It adds the correct UVs using the tile sheet we are using for textures.

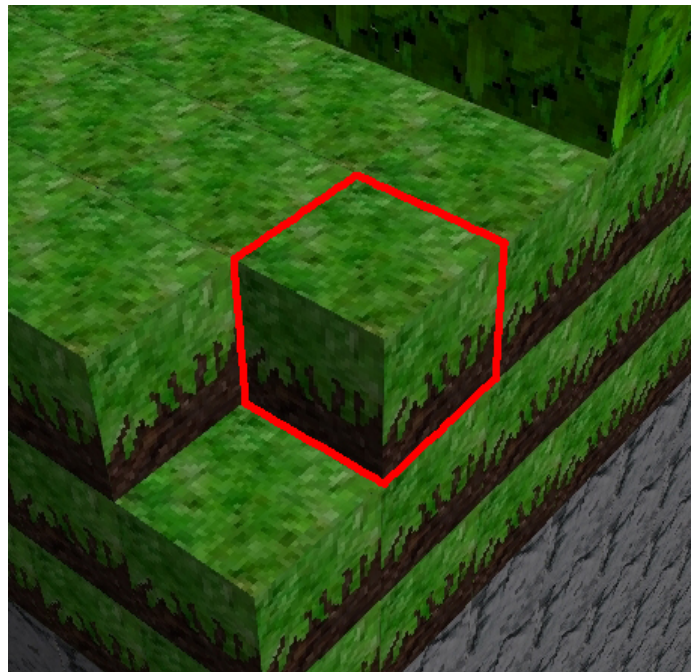


Figure 4.14: The Grass Block marked red only has 3 of the 6 faces rendered, because the other 3 are hidden and rendering them would just increase the computational overhead.

¹⁶For more information on Mesh Data, <https://docs.unity3d.com/2020.1/Documentation/ScriptReference/Mesh.MeshData.html>

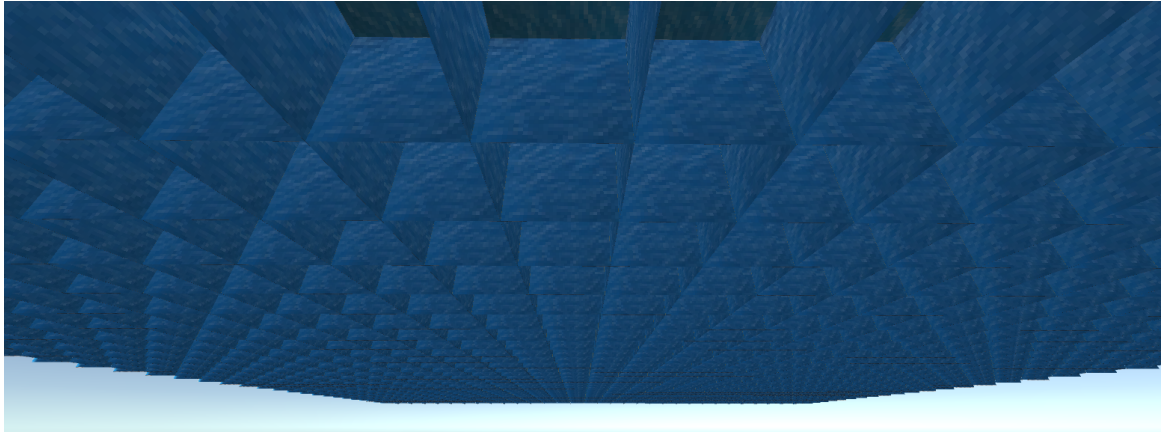


Figure 4.15: This screenshot was made with a view inside a terrain where meshes that cannot be seen from outside are being rendered. If they were not being rendered, this image would only show the sky box, which is what should be seen if we look to the terrain from the inside, meaning resources are being wasted to render useless meshes.

All the positions are cycled in each chunk, and all the blocks in every position are verified for their *Blockdata*, and this data is added to the *meshData*. This is the part where blocks that do not exist are checked again, either because the colour is not associated with any biome (because it was not recognized) or because the *Blockdata* of that block is non-existent. As said before, this is done so it does not stop our execution in the event of something unaccounted for happens, and if it does, a user can easily understand something is not right, just by looking at the terrain, which will have tall columns of stone for every block that does not exist. In the end, *UpdateChunk()* calls a final function which renders the mesh data we just gathered by sending it to the mesh and collision components associated with the instantiated Chunk object, Mesh Filter, Mesh Renderer and Mesh Collider, as we can see in Figure 4.8.

The next section will explain in greater detail how the blocks are handled, as well as the *Blockdata*, and how it is determined.

4.5.3.1 Blocks

Every block is essentially a subclasse from the class **Block**, which is the **Stone Block**. All subclasses have members that override the base class on the characteristics which change from block to block. One member is for the position of the texture of a certain block on the tile texture image and the other is a *bool* flag to tell if the block is solid or not, in other words, if the block needs collisions. Flowers or other plants are good examples of non-solid blocks. The main class has other functionalities, the most important being the *Blockdata* determining function.

In order to determine which faces are visible for every block, we need to verify which faces are adjacent to a solid block. The function basically verifies if, in any direction (north, south, east,

west, up and down) the following or previous block exists. So, for every position, we verify if the adjacent block on that axis exists, for example, when checking coordinates (x, y, z) , we go and check if there is a block on $(x + 1, y, z)$, $(x - 1, y, z)$, $(x, y + 1, z)$ and so on. Checking if a block is in fact on the position we are verifying is done by calling a *GetBlock* function, which simply takes the coordinate and checks if a block is in that position and if it is not, similarly to the *SetBlock*, it calls a function from the **TerrainManager** to check, with the given coordinates, in what chunk they are. The chunk verification is done by taking coordinates and comparing them to the chunk dictionary. If no chunk exists, it returns the **Air Block**, which is not solid, and the execution continues. If it exists, we simply verify if in those coordinates, inside the chunk we are in, actually exists a block, using the same function as before.

After all these verifications, every time it returns a non-solid block or the non-existence of one, it calls the respective function which adds to the *meshData* that specific face's vertices and UVs¹⁷, also using the block's tile texture which is already defined, as was previously mentioned. Table 4.2 will show all the blocks that can be found in this framework, its solidity and texture being used.

By now, we understand how the terrain is laid with the ordering the user desires and how the actual mesh is generated. The next section will detail how the vegetation, cave systems and overhangs are generated in the landscape.

4.5.4 Vegetation and Cave Systems

Vegetation and cave systems are actually laid at the same time the columns are being generated, in Section 4.5.2. It was not mentioned until now to ease up the comprehension, since we now understand how the blocks are placed, trees and caves are included in the terrain using the same notions.

First, important variables need to be set beforehand, in order to give caves and plants the same sense of generated randomness the rest of the terrain has, so a variable for the frequency of caves spawning was created, we will call it *caveFrequency*, as well as a size for these caves, *caveSize*. Depending on the biome, we created different variables for the frequency and size, for example, biomes which predominantly have mountains, will have a greater frequency of caves, as well as bigger caves in general. For the plants, we also need to setup some values, the *plantFrequency* and *plantDensity*, equivalent to the cave variables. The main difference for the plants is we are creating these two variables for each and every plant it exists in this framework, because how often a plant grows in a certain area, and how many plants of that species grow in that location, depends on the plant.

After setting up these values, we create a variable for the *caveChance*, which is determined by using our noise function already mentioned, with the scale parameter being the cave frequency.

¹⁷In Computer Graphics, the U is the horizontal axis and V the vertical axis, and they are coordinate system used for texture mapping. Each UV coordinate has a corresponding point in 3D space which is a vertice.

With this new variable, when we are at the point of comparing the layer of the block we are in to the *height* values we determined for each block, so we can place stone or dirt until a certain height, we add a new condition, to see if the cave chance on each position is greater or lesser than the size which was decided previously for caves in each biome. If it is greater, **Air Blocks** are placed instead, and this way we create our caves, and in mountain biomes, also overhangs.

The plants are created with the same premise, but each biome that has vegetation (all but the water and Alpine Tundra biomes) has a *bool* variable associated with it, as we seen in Figure 4.3, and has the job of controlling if a biome is generated or not with vegetation. This might be useful if the user wants, for example, to create a landscape with a beach and uses the Hot Desert biome to do so, but without the cacti and the shrubs. We use the values which were set up to determine the chance of spawning a plant for each position and we compare that value to the density we chose. The major difference is we are not only instantiating single blocks (like for the cave systems, we were instantiating **Air Blocks** when we wanted a cave), but we are also generating plants constituted of various blocks. Bushes and flowers are just a single block, but a tree for example, has two main parts to it, each part having several blocks integrating it, the trunk and the treetop. Every time a tree needs to be instantiated, a function which builds the tree is called. This function essentially cycles relative positions (not actual terrain coordinates) and places blocks, building each part of the plant. For the whole tree, for example, it uses four *for* loops, one for the trunk and other three for the leaves, and depending on the tree we want to generate, they will possess different generation methods.

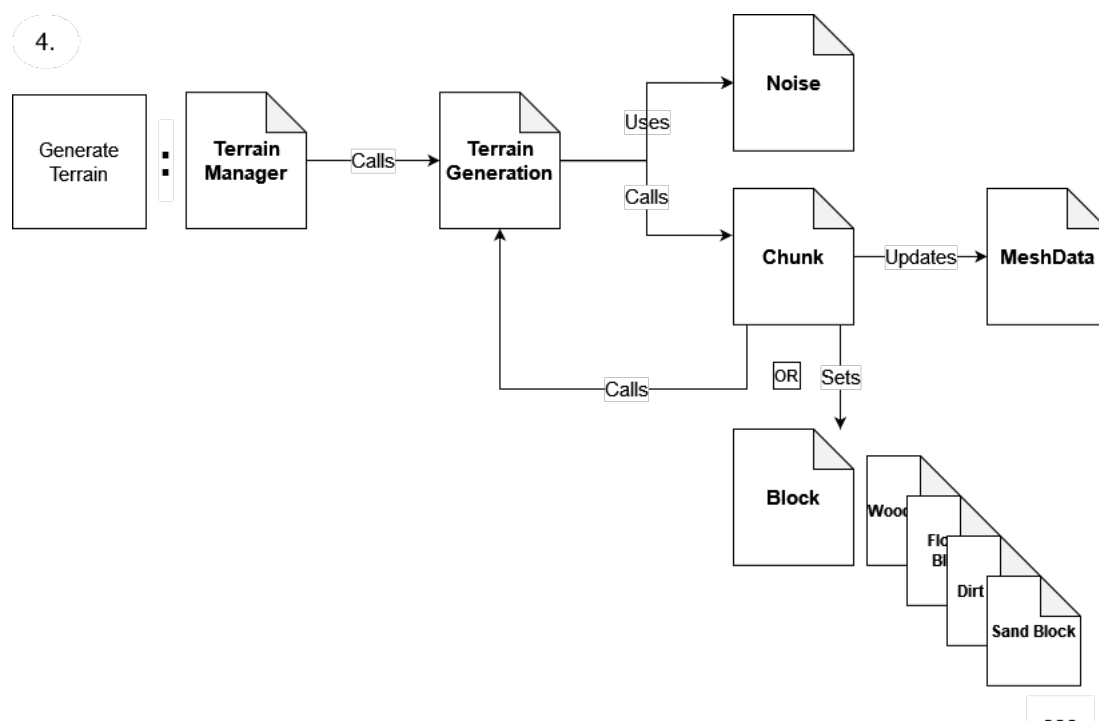


Figure 4.16: We can see the TerrainManager is the controller for all scripts

4.6 Cloud Generation

Clouds, even though they are not related to a terrain or its generation, are a feature which is always welcomed when dealing with landscapes, as they add another layer of immersion to the final look of the terrain, depending on the way we want to experience this landscape.

To start we had to create a Unity Material for the clouds, by first writing a very simple *shader* script¹⁸. The *Properties* block only has information on one shader variable for the colour white. We want the *Subshader* to render a transparent material and the *Pass* is the colour we defined earlier. The Material is then created using this shader.

The **Cloud** script starts by finding a reference to the *Player GameObject*, as well as determining the width of the texture (the texture in this case is Figure 3.27, as was previously said), the size of a tile of the texture, equal to the size of a terrain chunk and finally the height of the clouds. With this information, we call a function that loads the cloud data, which basically declares a *Color* array from our texture file. It finally declares and builds a *bool* array that varies depending on the color opacity.

From here, we create our clouds by adding to a cloud Dictionary the position of a cloud tile associated with the tile itself, created from another function which has the job of generating the cloud mesh.

Finally, the clouds are updated by changing their position with respect to the player position, on top of the terrain. Figure 4.17 shows the inspector of our cloud *GameObject*.

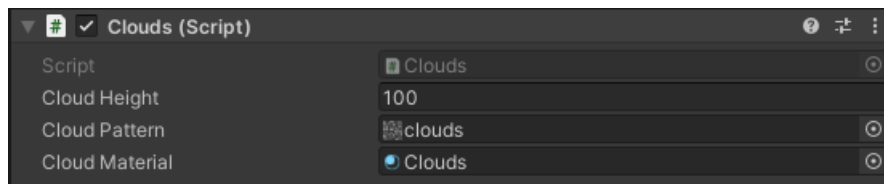


Figure 4.17: Cloud *GameObject* inspector, where we can define the clouds texture, material and height.

4.7 Terrain Visualisation

Terrain visualisation, as was said on the previous chapter is very important as is the way we have to assess the terrain we just generated. In this framework we have a script to control our cameras called **CameraController** and it is used as a manager to change between our camera views, each of them controlled by the respective script.

Three main cameras were included to visualise the landscape. The first one, which is the

¹⁸More information in <https://docs.unity3d.com/Manual/SL-VertexFragmentShaderExamples.html>

default view of the terrain, as soon as it is generated, is the orthographic camera. It is a simple camera, placed on a top-down perspective, pointed at an angle to our generated terrain, with a orthographic projection, which is a convenient way of visualising the terrain all at the same time, without moving the camera. Since we have 6 predetermined terrain sizes, the orthographic camera changes some settings to accommodate for this difference in terrain sizes, specifically its *Transform* component, in other words its position, as well as its size.

If the user decides to switch to a free camera, he can press the "K" key. It is also a simple camera, with classic free mode controllers, where the user can use the mouse to rotate the camera and the "W" and "S" keys to go forward and backward.

Finally, if the user presses the "L" key, it changes to our player view. This view is essentially emulating a player in our terrain, affected by physics and real world constraints, so it has public player controller variables to give a sense of reality, like the walking speed, mouse sensitivity, jump height, gravity acceleration value, and others. The inspector view of the Player *GameObject* is shown in Figure 4.18.

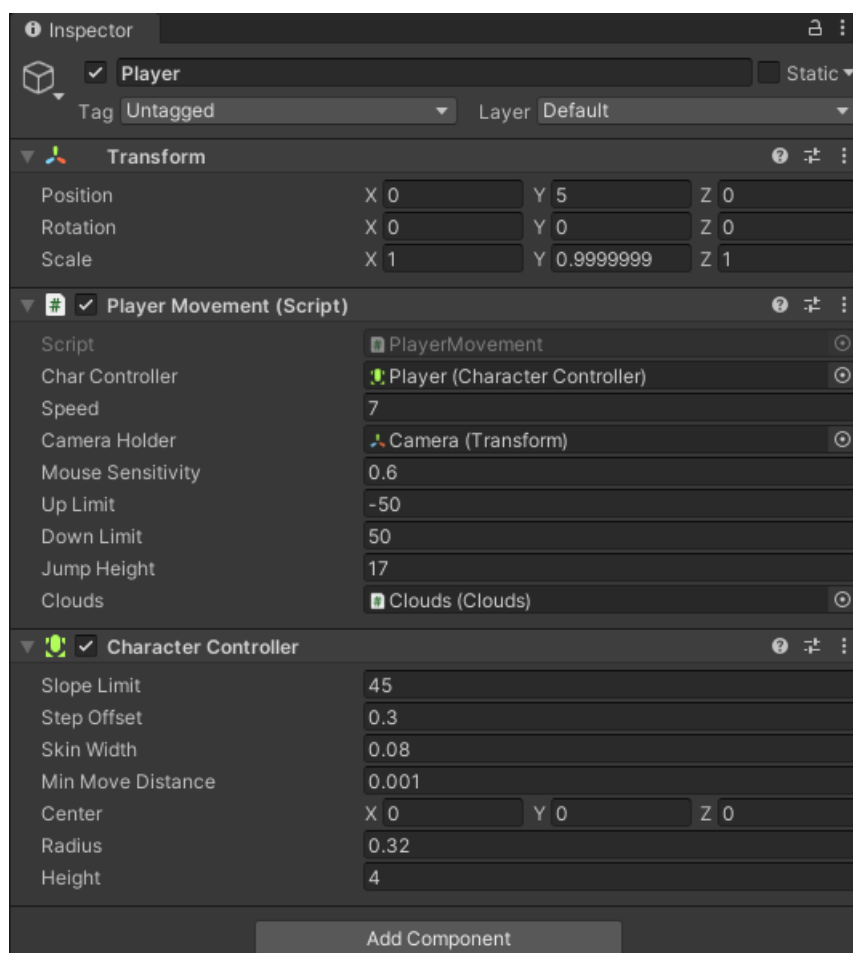


Figure 4.18: Here we can see the values assigned to each controller variable for the Player, such as the speed, up and down limits, jump height, reference to cloud script and others

The **PlayerMovement** script is pretty straightforward, as it basically has two functions for the movement, one to move around and the other to rotate, using the mouse. These two functions are placed inside the *Update()* loop and we have our player movement controller complete. These functions are done in the most basic way possible: we just need a player in our terrain to experience it from the player perspective, and so, the player controls do not need to be the most refined and do not need to have a lot of features. The player moves by updating its position from the Input keys that represent the "Horizontal" and "Vertical" axis. Rotation is done in a similar way, but this time using input from the mouse. This way, pressing the horizontal and vertical movement keys, "W", "A", "S", "D", while looking around with the mouse controls our player. The space bar is used to jump.

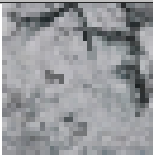
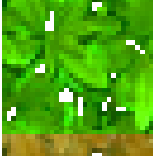
4.8 Conclusion

This section aimed to give a cohesive and logical explanation of how this framework was achieved, detailing every aspect of the sketch phase, generation procedures and additional features.

We started by showing a general sequence of events with the help of a diagram. We then followed the diagram logic sequence and detailed each phase's implementation, starting from the sketch UI, how it is used and how it was built. After it was given the explanation of how the sketch is interpreted and what information is extracted and used to later determine the terrain layout and further generation, and how the noise algorithm affects it. To finish the chapter, it was explained how the clouds are generated and how they react to the player moving on the terrain, as well as how the other two camera perspectives are used and how they were achieved.

By now, the reader should have a good understanding of how the whole framework works and how it was implemented.

Table 4.2: Tiles used for each block present in this framework with its respective solidity

	Solidity	Tiles		
Stone	Yes			
Air	No	-		
Normal Grass	Yes			
Tropical Grass	Yes			
Dry Grass	Yes			
Dark Grass	Yes			
Normal Leaf	No			
Tropical Leaf	No			
Dry Leaf	No			
Dark Leaf	No			
Normal Wood	Yes			
Tropical Wood	Yes			
Dry Wood	Yes			

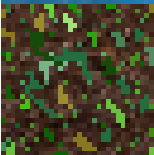
	Solidity	Tiles
Sand	Yes	
Snow	Yes	
Water	No	
Dirt Grass	Yes	
Blue Flower	No	
Red Flower	No	
Pink Flower	No	
Cactus	Yes	
Dry Bush	No	
Frozen Bush	No	

Table 4.3: Water Masses Base Values

	WaterBaseHeight	WaterNoise	WaterNoiseHeight
Water Masses	3	0.0001	2

Table 4.4: Hot Desert Base Values

	SandBaseHeight	SandNoise	SandNoiseHeight
Hot Desert	3	0.04	3

Table 4.5: Cold Desert Base Values

	SnowBaseHeight	SnowNoise	SnowNoiseHeight
Cold Desert	3	0.04	3

Table 4.6: Taiga Base Values

	DirtBaseHeight	DirtNoise	DirtNoiseHeight
Taiga	1	0.04	3

Table 4.7: Alpine Tundra Base Values

	DirtBaseHeight	DirtNoise	DirtNoiseHeight
Alpine Tundra	1	0.04	3

Table 4.8: Temperate Forest Base Values

	DirtBaseHeight	DirtNoise	DirtNoiseHeight
Temperate Forest	3	0.04	3

Table 4.9: Tropical Forest Base Values

	DirtBaseHeight	DirtNoise	DirtNoiseHeight
Tropical Forest	3	0.04	3

Table 4.10: Savannah Base Values

	DirtBaseHeight	DirtNoise	DirtNoiseHeight
Water Masses	3	0.04	3

Chapter 5

Results and Validation

The contents of this chapter will cover the visualization of the framework's results and further evaluation and validation using the adequate tests that corroborate the correctness of the solution. The applied tests should visually validate the most important features of the framework such as the terrain coherence that is trying to be replicated from real life examples, the uniqueness of the generated terrain and the framework's expandability and versatility to accommodate the addition of more of the existing features or new ones to be implemented.

5.1 Terrain Results

This section will show the final results of the generated biomes the framework produces. To properly evaluate and visualize the results of each biome independently, terrains containing only one biome will be generated using different sketches painted with each colour available. We will demonstrate the terrain using the orthogonal camera view and an additional viewing angle using one of the other available methods. In the end, the result of the sketch in Figure 3.2 will also be presented to show results of a terrain using all the available biomes. The terrain sizes will be $[128 * 128]$ for the following demonstrations, using a seed value of 0.

It is important to note that the biomes that were integrated in this framework are represented in the way we consider them to be, accommodating the features that were envisioned by us when searching for real life data as a reference, because in real world conditions the same biome can have different topographies and characteristics depending on their location.

5.1.1 Water Masses

Water Masses are the regions that are the easiest to emulate, since everything that they try to accomplish is to be a flat plane composed of water blocks, with low height, in order for the water level to be below all other biomes and thus be used to create any form of water formation, either

being rivers, oceans or lakes.

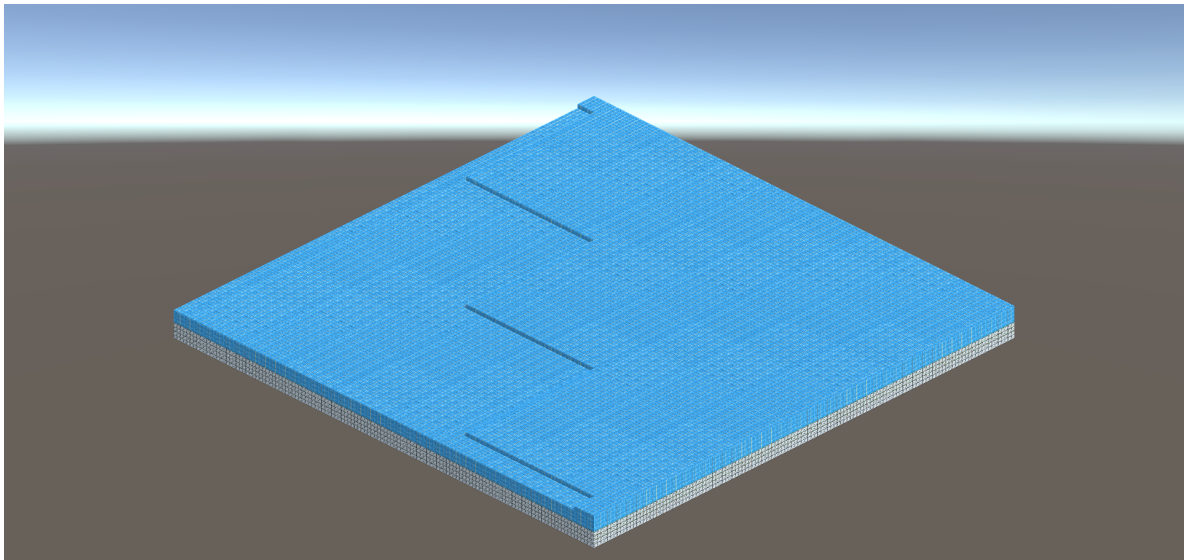


Figure 5.1: Water Biome

5.1.2 Hot and Arid Desert

Regarding the generation of hot deserts, it was considered important for them to be mostly flat but irregular and incorporate sporadic shifts on the terrain height to represent dunes, which can be small or larger. Cacti and dry bushes were added to give more of a desert feel, as these are the most common forms of vegetation one can find in this biome. Small cave formations can also be found scattered but very sporadically, because most are covered in sand.

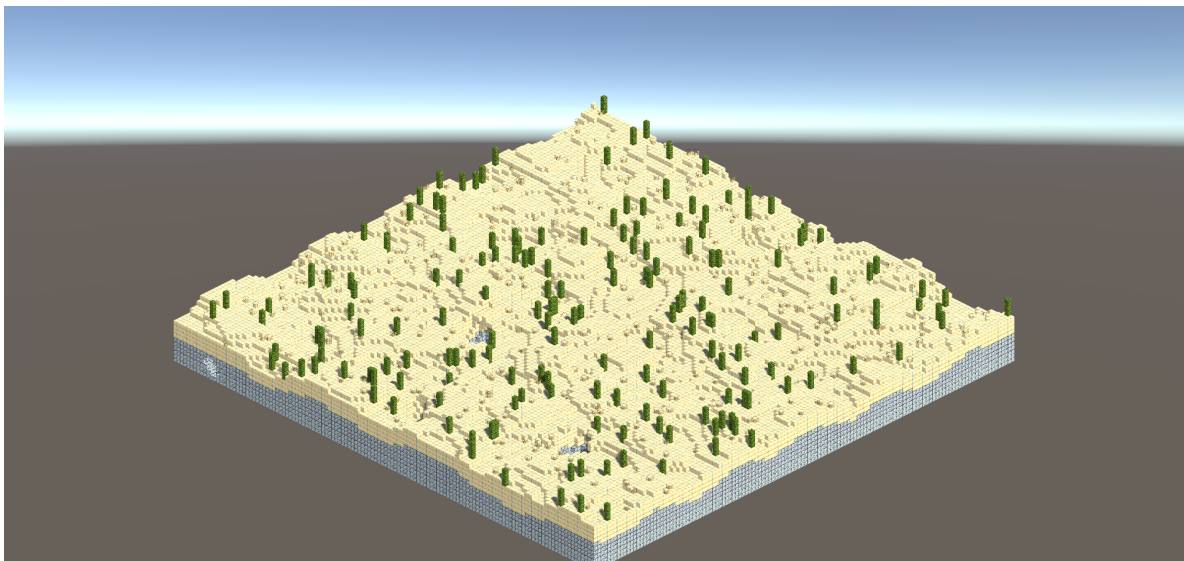


Figure 5.2: Hot Desert Biome

Figure 5.3 demonstrates the vegetation that populates the landscape, as well as a larger cave that can be seen in the sand.

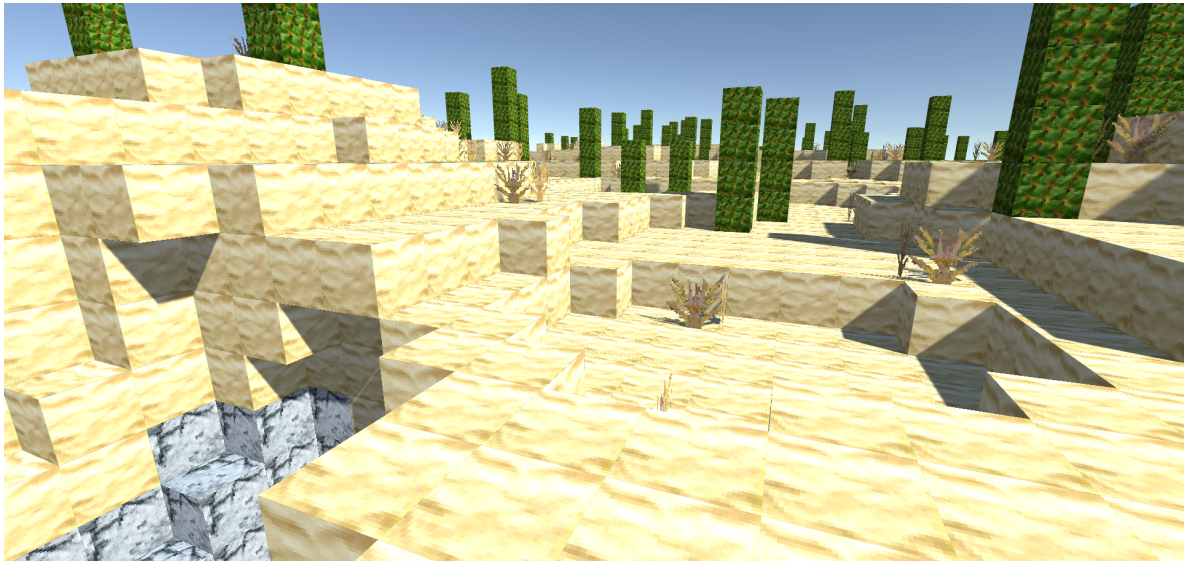


Figure 5.3: First Person View of the Desert Biome

5.1.3 Cold Desert

Cold desert generation is very similar to the hot deserts, as both try to represent an inhospitable place with low vegetation and flat topographies. Cold deserts are even flatter than their hot counterpart, and possess less vegetation, only including some sporadic frozen bushes. The intention of creating a frozen wasteland was accomplished when comparing to the used references, such as Figure 3.7.

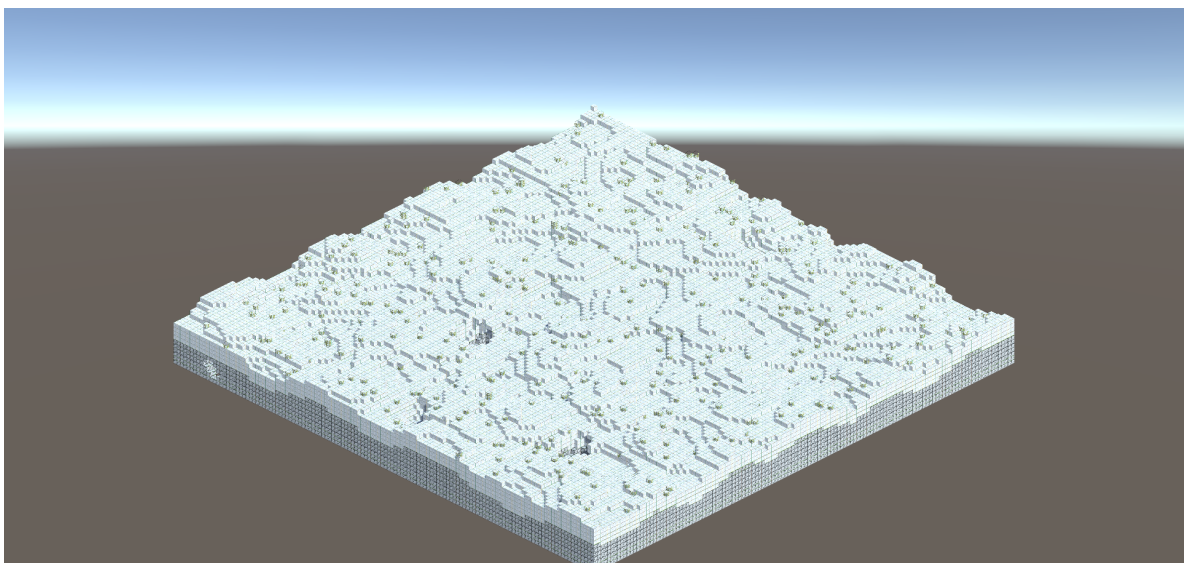


Figure 5.4: Cold Desert Biome

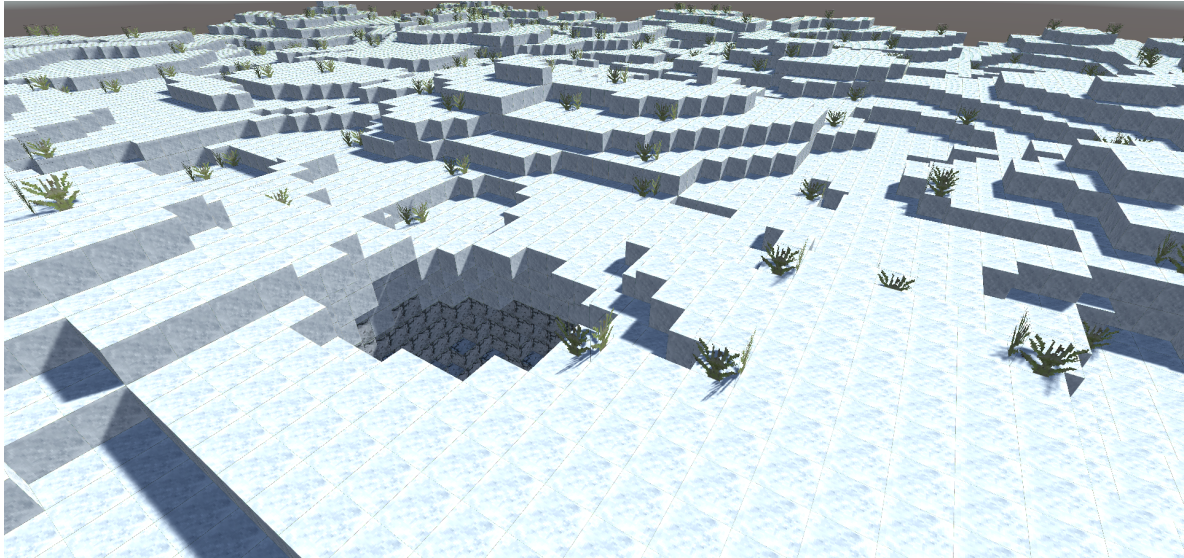


Figure 5.5: First Person View of the Cold Desert Biome

5.1.4 Temperate Forest

Temperate Forests thrive with vegetation and are dominated by trees, bushes and ground vegetation such as flowers. To recreate this, three different types of flowers were scattered across the ground, as well as some bushes. The ground is mostly populated by oaks, where in some areas the sun light is blocked, due to the dense treetops. Temperate forests are usually dominated by flat areas but some hills and terrain elevations can also be noted as well as cave entrances.

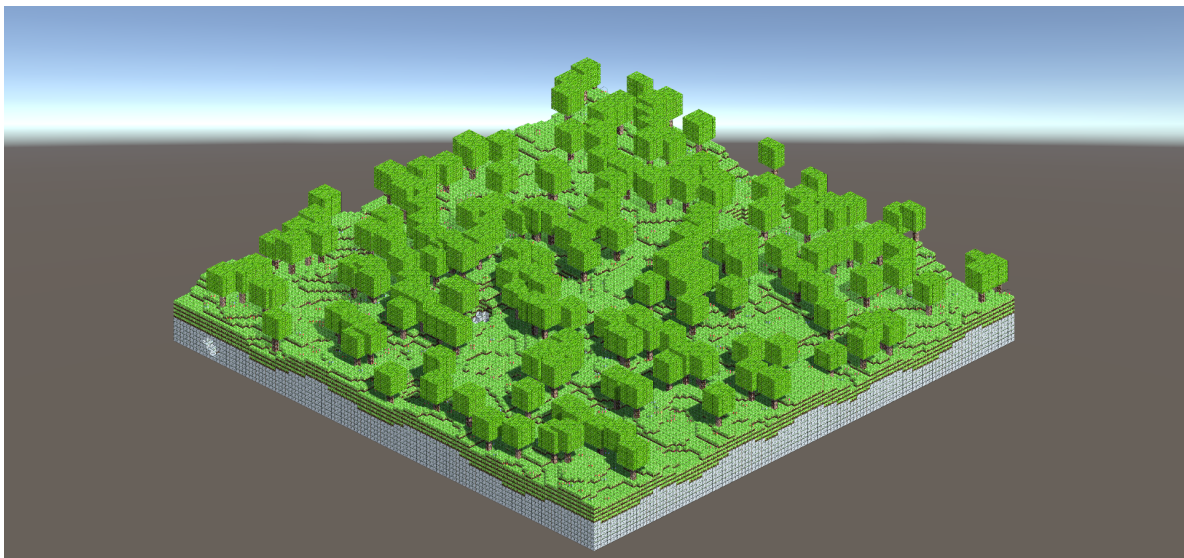


Figure 5.6: Temperate Forest Biome

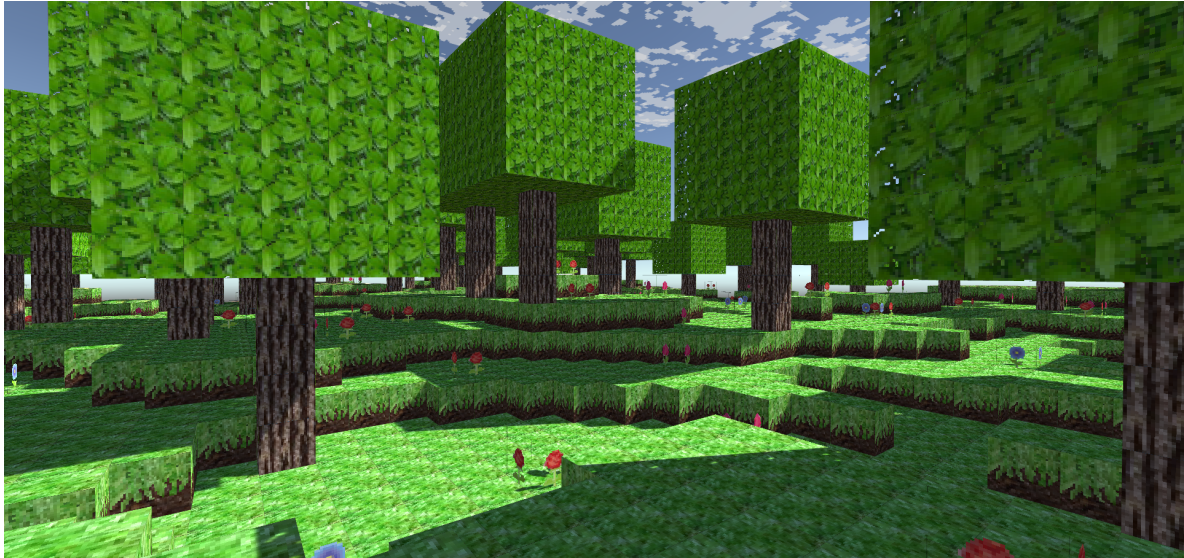


Figure 5.7: First Person View of the Temperate Forest Biome

5.1.5 Taiga

Taigas are very irregular with tall mountains or low valleys so noise values were chosen accordingly to emulate this topography. On the tallest mountains we can find snow and the landscape is populated with tall pine trees, having references like Figure 3.9 in mind. The mountains sometimes spawn with structures like overhangs or can have cave entrances to the inside of the mountain, which in this biome are larger than less mountainous biomes.

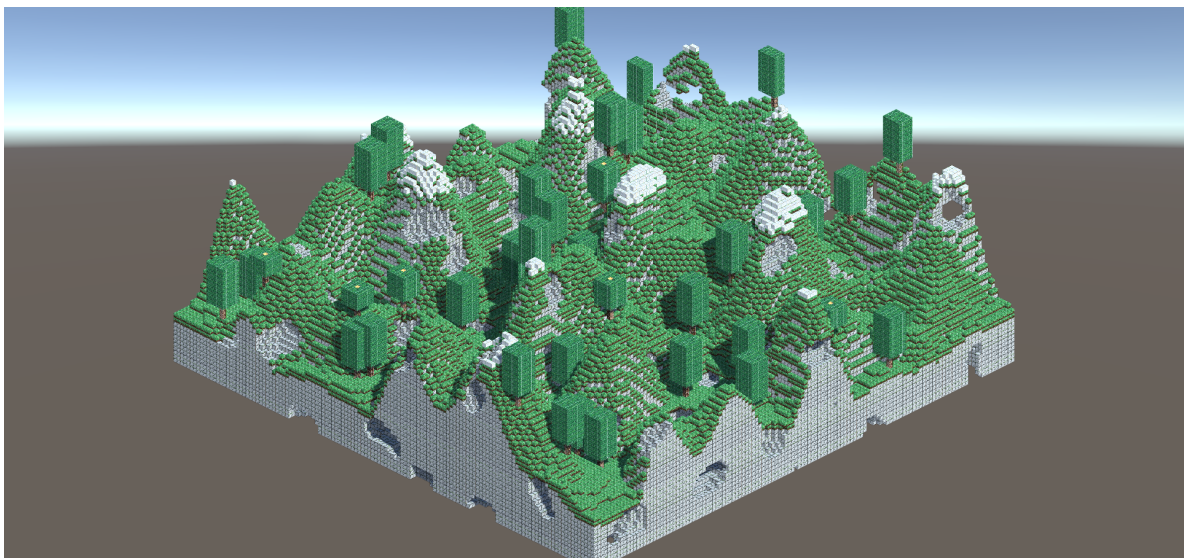


Figure 5.8: Taiga Biome

In Figure 5.9 can be seen one overhang on the side of a mountain.

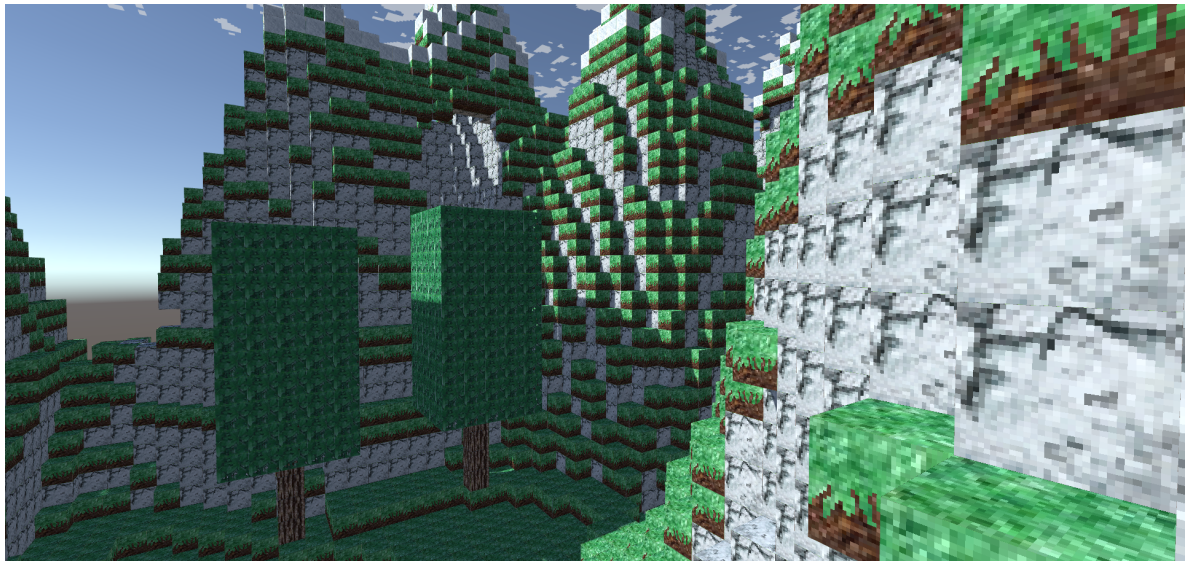


Figure 5.9: First Person View of the Taiga Biome

5.1.6 Tropical Forest

The most dense vegetation can be found in Tropical Forests, with tall ceiba trees and compact bushes and ground vegetation that makes this biome difficult to traverse on foot. It has similar topography to a Temperate Forest but most of the cave entrances are completely covered with vegetation, as can be seen in Figure 5.11

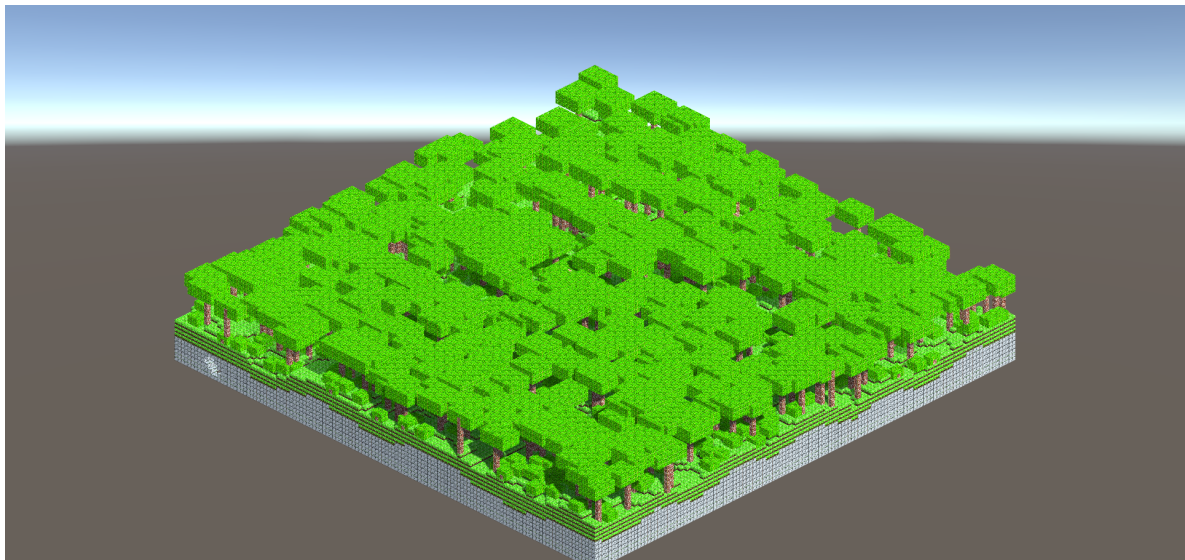


Figure 5.10: Tropical Forest Biome

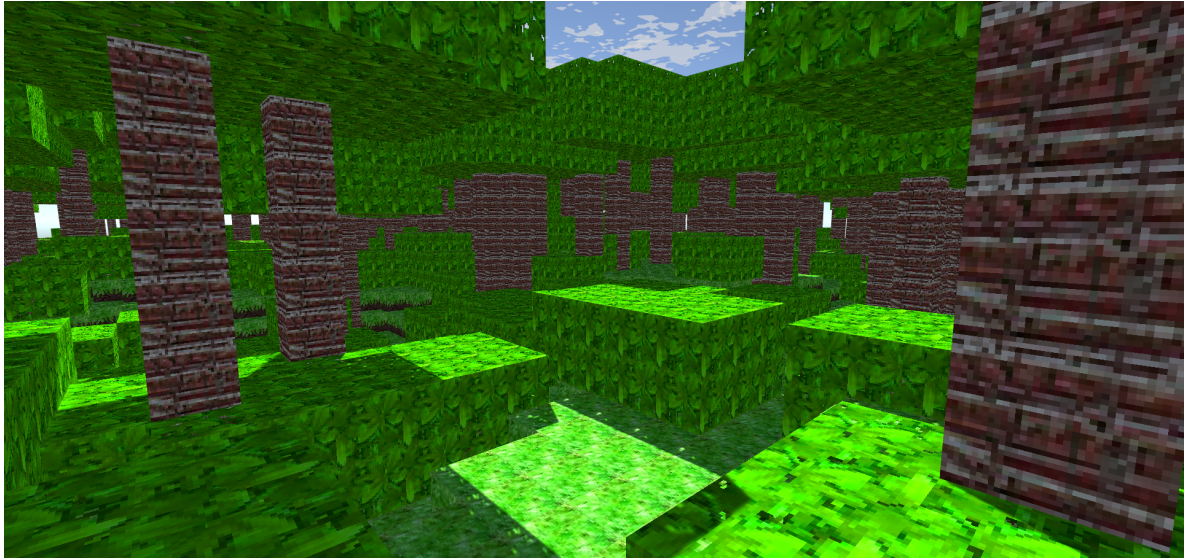


Figure 5.11: First Person View of the Tropical Forest Biome

5.1.7 Alpine Tundra

Another very mountainous region that due to its harsh weather makes it difficult for even grass to grow, so trees were not included in this biome. A lot of rock formations with overhangs can be found as well as caves and holes. The top of the tallest mountains have snow.

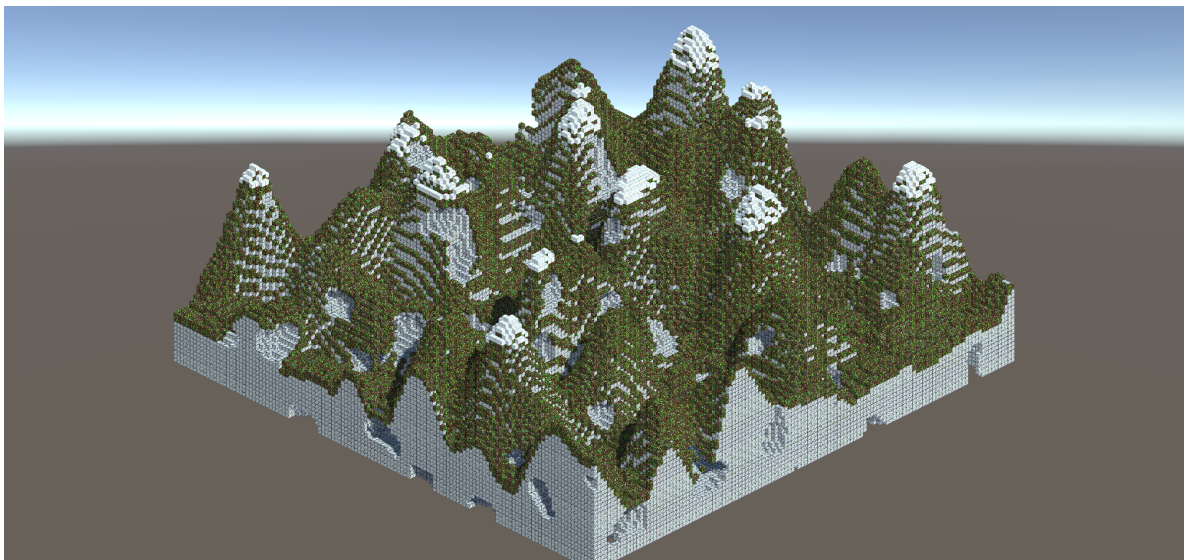


Figure 5.12: Alpine Tundra Biome

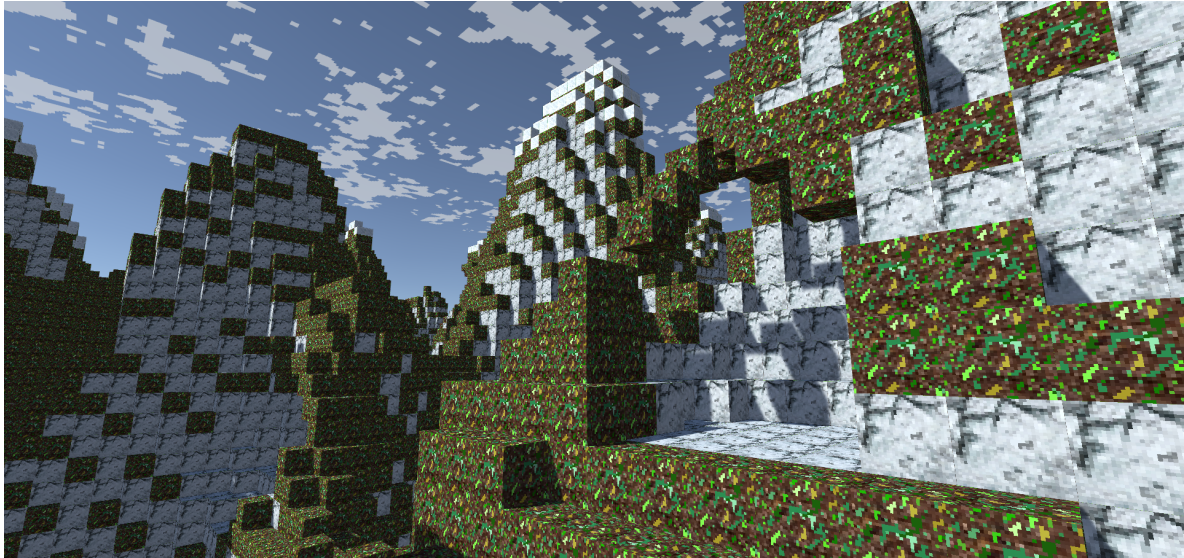


Figure 5.13: First Person View of the Alpine Tundra Biome

5.1.8 Savannah

The savannah is very flat and dry, with limited vegetation, only some trees stand out, as demonstrated in Figure 3.12.

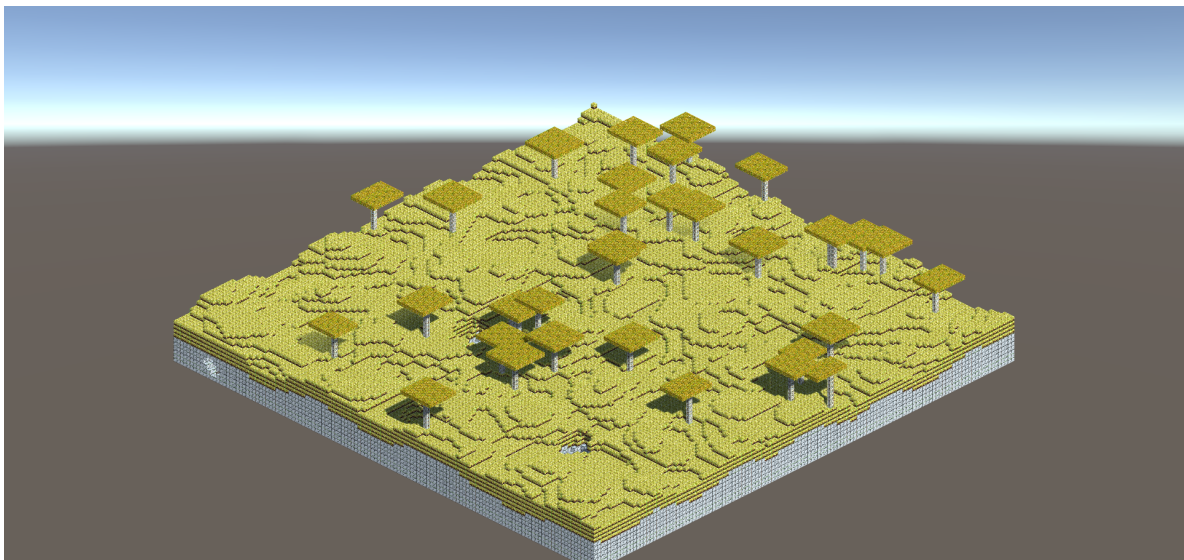


Figure 5.14: Savannah Biome

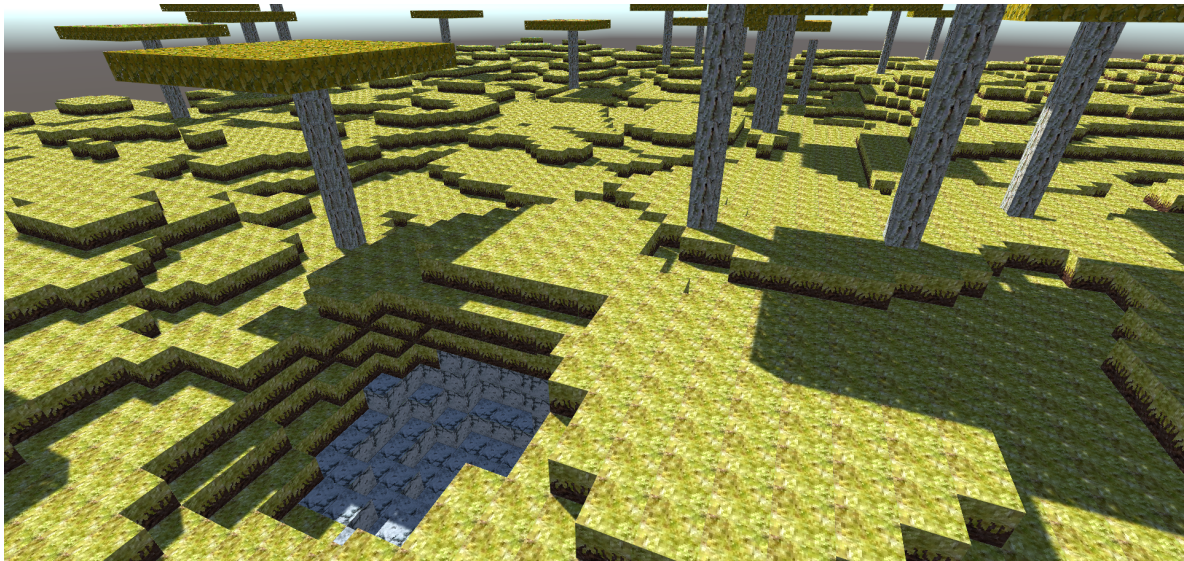


Figure 5.15: First Person View of the Savannah Biome

5.1.9 Combination of all biomes

The combination of all biomes in a single image can produce interesting and varied landscapes and environments. This section shows what the terrain from the sketch in Figure 3.2 looks like. It is a $[256 * 256]$ surface, where everything is much more detailed. We first have an orthogonal view of the terrain in Figure 5.16. Figure 5.17 is a good way to compare the sketch that was made to actual terrain. Figure 5.18 gives a sense of what the landscape looks like from inside the terrain, where we can also see the generated clouds in the sky.

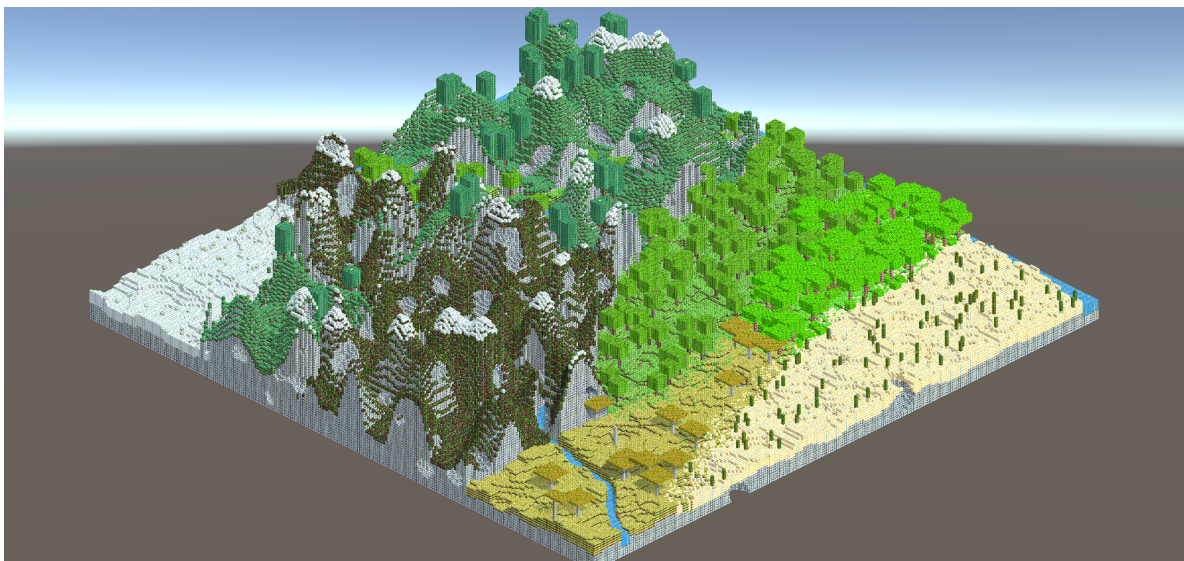


Figure 5.16: Orthogonal view of the example sketch



Figure 5.17: Top View of the generated terrain from the example sketch



Figure 5.18: View from inside the terrain, from a first person perspective

5.2 Terrain Evaluation and Validation

In this section we will cover the main pre-requisites we established before the start of this thesis, and we will evaluate if these requisites were actually delivered and to what degree they actually fulfill the intended goal.

5.2.1 Validation on Terrain Size Coherence

A very important requisite to validate is the size of the terrain, and how the change in size will affect the look of the world.

The framework is prepared to generate six different pre-sets for the terrain size from which the user can choose but in reality, the framework has the capability of creating a terrain of any size the user wants, provided the adequate sketch and respecting the already mentioned conditions. The user also needs to have in mind that this framework is not in any way optimised to generate larger terrains very fast: Table 5.1 shows execution times for every available terrain size, using the most simple variation, the water biome. This test was made using a machine with the following specifications:

- CPU: Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz
- GPU: NVIDIA GeForce GTX 1070 Ti
- RAM: 64GB

We can clearly see that, obviously, the larger the terrain, the longer it takes to generate, but the time interval we wait is exponentially higher, every time the terrain size is increased. Also, what is considered "large" or "small" depends on the user, so if he wants a terrain even bigger than the maximum we have set, the execution time should take much longer.

Table 5.1: Execution times, in seconds, for the generation of the water biome in each terrain size.

	Time
[32 * 32]	1.853041s
[64 * 64]	2.294375s
[128 * 128]	11.81499s
[256 * 256]	148.8619s
[512 * 512]	2386.674s
[1024 * 1024]	>7200s

Since a voxel in this framework, has [1 * 1 * 1] in Unity's units, and comparing terrains generated with a larger size against terrains with smaller size, even though the latter are generated much

quicker, they are not as interesting when compared, because the general size of the terrain is smaller but each single unit is the same, resulting in much less detail in the terrain and making the biomes' borders appear more drastic in the way they transition, since it uses less blocks to do the same thing. This can be clearly observed when comparing the three terrain sizes, the smaller one being the less coherent, and in terrains like Figure 5.19 we can note that the Taiga biome surrounding the whole terrain does not look much like a mountainous region and more like a wall, since the painted area in the sketch only covered that amount of pixels, which translated to the smaller number of blocks which compose the "wall".

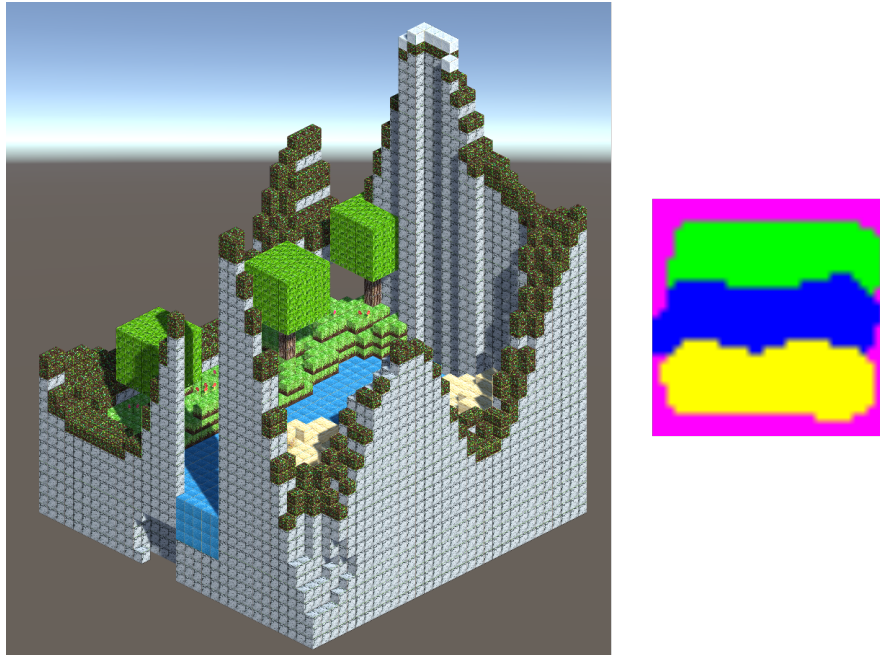


Figure 5.19: On the sketch phase, the borders were painted magenta, resulting on a "wall" around the terrain, and not mountains, due to the lack of space a $[32 * 32]$ terrain inherently has

Terrains with size $[128 * 128]$ and $[256 * 256]$ are a good compromise of speed and consistency because the generated terrain has reasonable details, as we can see in Figures 5.16, 5.17 and 5.18, since the terrain starts to possess an adequate size to make it look more coherent and does not take much long to be generated. With this in mind we can clearly observe in Figure 5.21, a terrain generated using the sketch in Figure 5.20, that the best looking terrains this framework can produce are the larger ones, the larger the better, since they have enough space to accommodate any details that a user may want.

A good solution to this issue would be to implement a way to control the scale of each voxel, while maintaining everything else the same, and coherent.

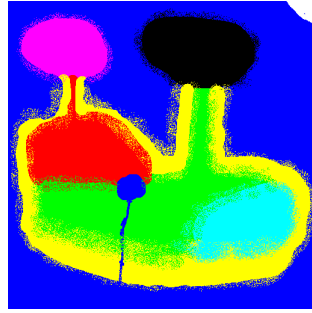


Figure 5.20: Sketch of a $[512 * 512]$ terrain

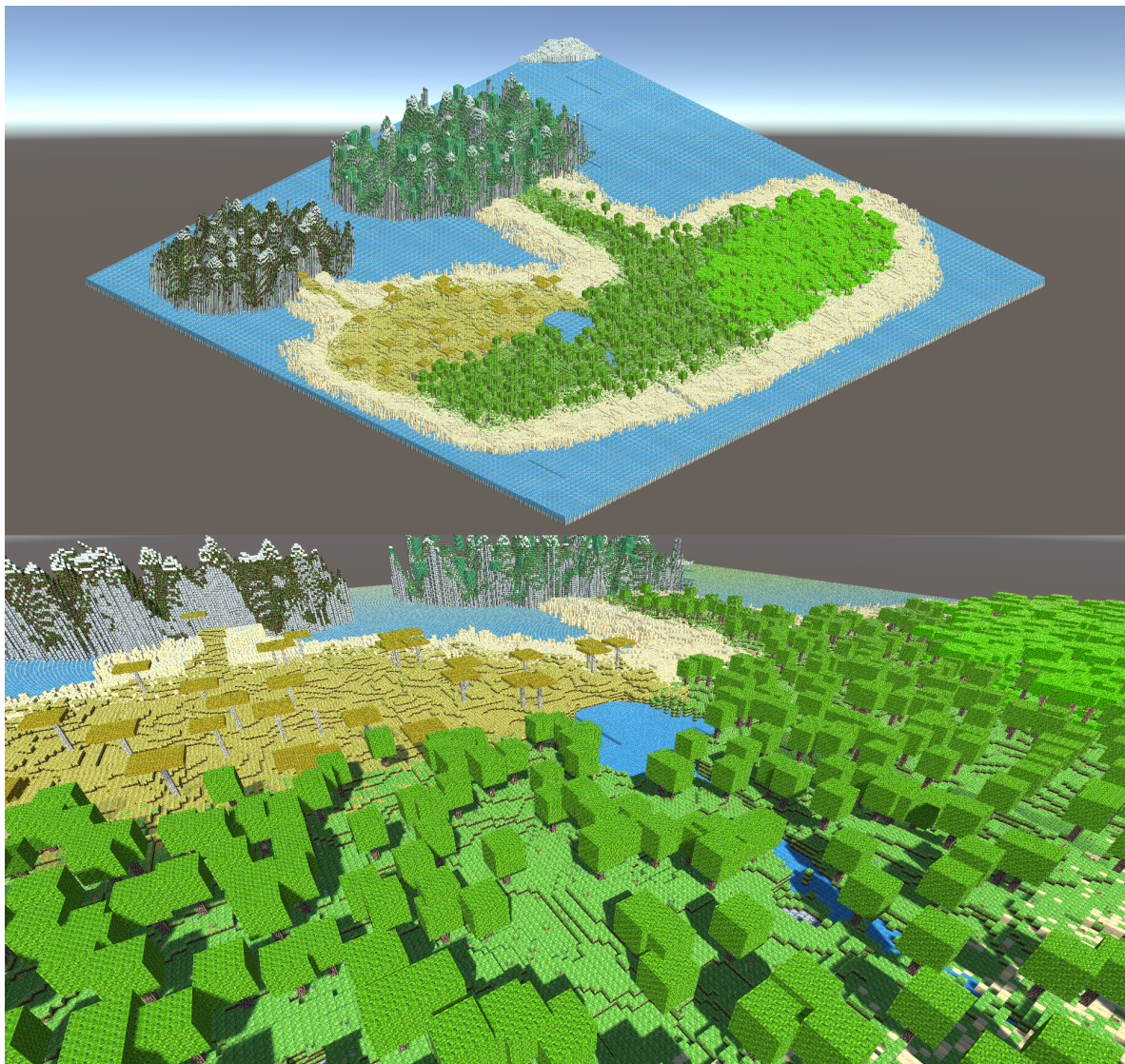


Figure 5.21: A terrain with the size $[512 * 512]$, the second biggest the framework is prepared to output without modifications. We can notice how the increase in size allows for better and finer detail, since our basic unit is now much smaller in proportion to our terrain as a whole.

5.2.2 Validation on Terrain Uniqueness

One of the goals of this framework was for it to have the possibility to generate completely unique terrains from user to user. Said uniqueness first comes from the fact that giving a user a blank canvas that can be coloured in any way with 8 different colour choices that translate to 8 different biomes, will always result in completely different sketches. Also, if the user has the option to turn off vegetation generation for any specific biome, it can be used as one more differentiating factor. This user defined sketch combined with the parameter *seed* which was implemented, results in an endless amount of variations in the terrain features for the same sketch. In Figures 5.22 and 5.23 we can observe the terrain created from a sketch, using a *seed* value of 0 (top image) compared to a *seed* value of 10 (bottom image). Even though it is the same sketch, the biomes have different characteristics that will be compared.

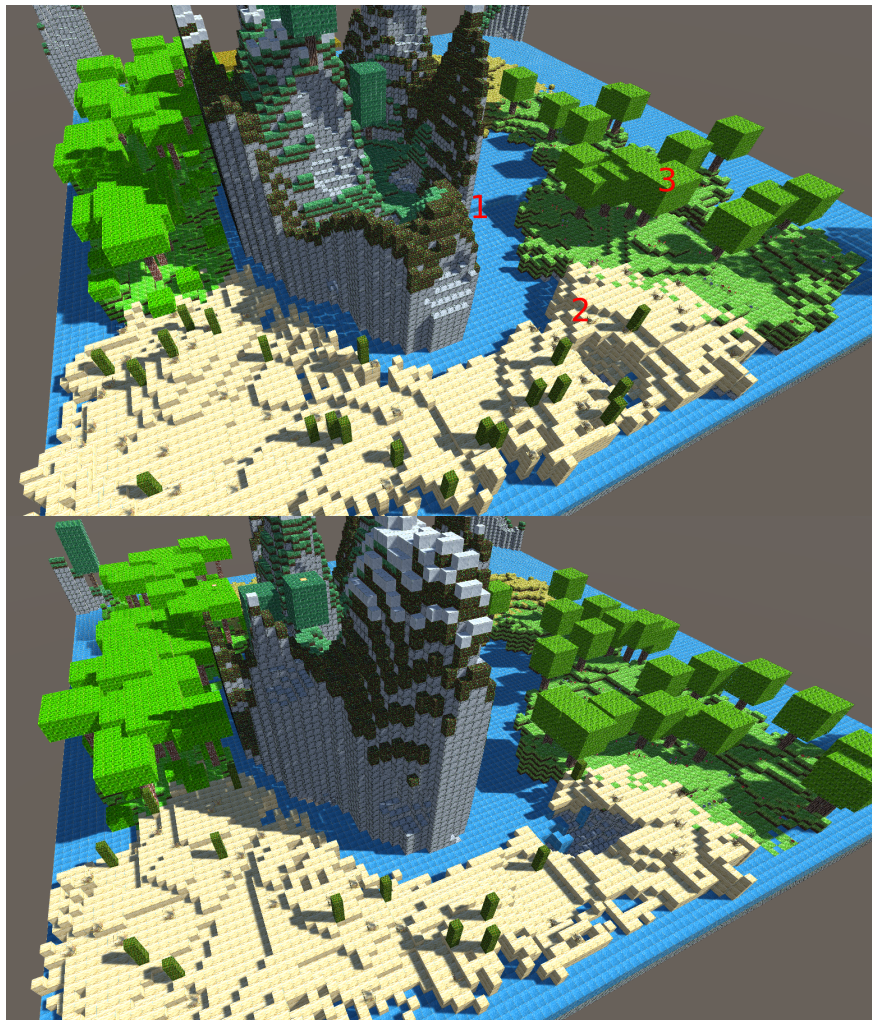


Figure 5.22: (1) - We can see the difference in *seeds* result in the Alpine Tundra biome producing a much shorter mountain on the same location compared to the bottom. (2) - We can notice even new cave systems can spawn depending on the *seed*. (3) - Vegetation spawns in different locations, notice how even the number of cacti changed between *seeds*.

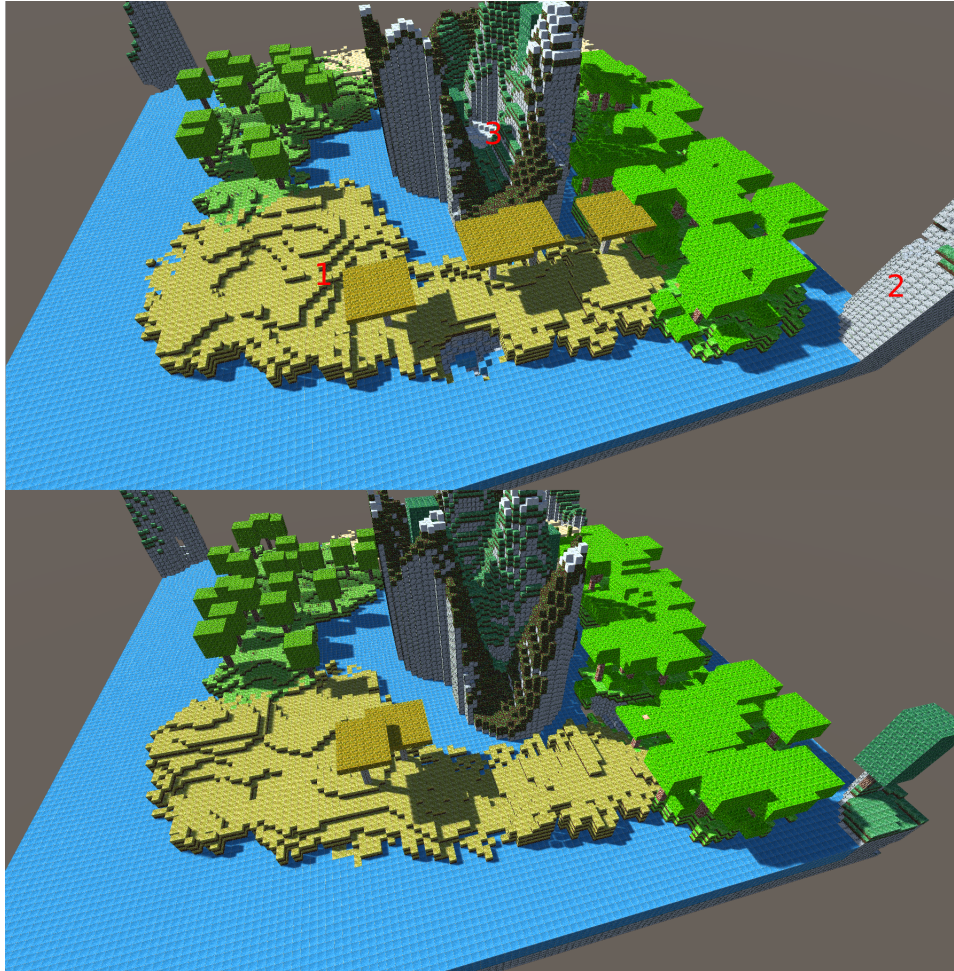


Figure 5.23: (1) - In this Savannah biome we can clearly notice how different seeds produce different topographies, as not only the tree positions differ, but the ground is generated distinctly. (2) - The distinct *seed* values originate a tall mountain opposed to a low height surface, even with a tree on the bottom. (3) - We can see the center Cold Desert opposed to the bottom image.

As was previously mentioned during the implementation phase, one other feature that contributes to the uniqueness of a terrain, is to give the user the option to decide if he wants a terrain to be generated without vegetation. This gives the possibility to use certain biomes as other options. Figure 5.21 shows how the Hot Desert biome was used to make a beach around all the land, because it was generated without cacti and shrubs.

5.2.3 Validation on Framework Expandability and Versatility

A very important goal to achieve with this framework was for it to be completely expandable to accommodate any new features and be versatile, so it can be used as a base for any project that might find a terrain generator from sketch useful for its goal.

Everything in the framework can be expandable. We can add different blocks, since every block

is a script with a subclass of the class **Block**, it is easy to create more scripts to include different blocks. More and different vegetation can be added easily, since every plant that appears in the framework is either a single block (which just needs to be set in the scene) or a combination of different blocks, like trees for example, as we saw in Section 4.5.4, which the blocks need to be positioned in the correct places in order to coherently create a shape of the plant we are trying to add. The number of biomes also can be expanded to any number of biomes desired, since we just need to add a new colour to the image reader and make the necessary changes for the positions of that colour to be saved and add a new function to place the column of blocks for each position, as was explained in Section 4.5.2, adjusting noise values for the biome's topography and setting the correct blocks.

The framework can be easily used as a base for a new application or game, by refining mechanics like the character controller for movement and to make it interact with the terrain, or by removing the ones that exist and add new ones. It can be transformed to include massive terrains, or even infinite ones (by repeating sequentially the user sketch, for example) without causing much of a computational burden. Since the terrain was created to be generated in chunks, already having in mind the possibility of having to create and destroy these chunks dynamically, as our player moves through the generated world.

To change the look of the generated terrain the tile map can be easily replaced with one that fits the goal of the application more, letting the look of the terrain be completely to the taste of the person who is developing using this framework as a base.

Since the framework was made in a modular way, completely new additions can easily be made, for example including a day and night cycle, a weather system or even animals and creatures. It is in the hands of the user to expand it if he so desires.

5.2.4 Validation of the Accuracy on the Realism of the Terrain

The last requisite we made, was for the terrain to have a degree of reality, so it can be used in projects where the goal is to be as accurate as possible to the real world.

During the development of this work, the way we deal with this requisite changed and we now feel it is the requisite more susceptible to a negative evaluation. To start, the idea of "realistic environment" clashes with the fact that the user can completely decide the ordering of the terrain, giving him the possibility of having a really hot environment right next to a cold one, which is something which does not have any real world coherence. Next, during the development of this work, it was decided to change the look of the terrain to a voxel block world. This results in terrains that are obviously completely unrealistic compared to real world terrains, which are most definitely not made of blocks. The tile map that was originally chosen tries to tackle this problem to some degree, since it uses photos and images of real world components, like dirt, grass, stone, and others. Obviously this is not a fix, since the blocks will always be composing the terrain. However, having the landscape be generated with blocks, opens possibility to other

forms of media with this look in mind, so generating using our approach is not a deal breaker and can have various uses.

Another aspect of the way we generate the terrain is the fact that even though we are using blocks, this is the foundation for using an algorithm called the Marching Cubes, since this algorithm has a 3D grid organization as a base, exactly as it was made in this framework. Generating the terrain using this algorithm would give it a much more realistic look, while still maintaining the same user sketch functionalities, because even though it may look like the terrains generated this way have nothing to do with blocks, they are organized as blocks. This approach to the Marching Cubes algorithm will be detailed in the next chapter.

5.3 Conclusion

In conclusion, having in mind what was discussed during the present chapter, we can say that, to our standards, the pre-requisites we established, were achieved. A coherent terrain generator was created, which is completely driven by the user premises and decisions, and thanks to the sketch phase allied to a *seed* value also chosen by the user, it produces completely unique terrains. Even though these terrains are not the most visually representative of reality, they can have any size and have the possibility to be expanded with new features, functionalities, mechanics or even tweak the terrain generation values to better suit any needs.

The next and final chapter will conclude this dissertation as well as mention future work related to the framework.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

During the development of this thesis and framework, we were able to accomplish the goals we set in the beginning. We first made a revision on the State of the Art methods and techniques on Procedural Generation of landscapes, terrain features and plants.

Next, the goal was to find an answer for the questions that rose when thinking about the solutions we should implement to accomplish the goal of creating a terrain generator that takes a user sketch and uses it to create a landscape ordered as the user wanted, using procedural generation methods. What the sketch layout is, what biomes should be implemented, how to implement the features of each biome, how to make this framework expandable and versatile to someone who wants to use it to build another framework or application. A great amount of research was needed to understand what were the best choices we could use, in order to achieve the goals we had. Some aspects had to be reworked during the implementation, and it resulted in the restart of the work, as said in Section 3.6.

Most of the parameters used regarding the generation of different features, like caves and vegetation were decided on trial and error until it had the look we were aiming for, depending on the biome we were trying to emulate. This kind of trial error in search of the ideal parameters is very time consuming and can not really be done any other way, as we are searching for the parameters that will have a good looking end result, which is highly subjective. As such, we present a framework well suited for a user that needs a terrain built to his needs, as well as a developer who needs the terrain generation from a sketch aspect to other uses.

The code for the implementation of this framework is available online at: <https://github.com/andreministro/TerrainGeneratorFromUserSketch>.

6.2 Future Improvements

This terrain generation framework, produces landscapes with a variety of possible biomes with different and unique topographies and features, very well suited for a range of uses, due to the versatility the sketch phase provides. As was mentioned on the previous chapter some improvements and reworks can be done in order to add more value and a better degree of detail and reality to this project. The next sections will present some ideas for future improvements and some insight on the potential implementation.

6.2.1 Implementation of an Ecosystem

Since this framework already includes plants (and has the possibility to include more different plants), all dedicated to the specific biome they are found, instanced using procedural methods, it would give a new degree of immersion to include animals native to each environment, that would interact between each other and with the plants.

Implementation of wildlife on every biome would require a library of 3D models of animals to be procedurally instanced. Obviously this is the beginning and the most basic place to start the addition of this feature. In order for it to be a coherent environment, it would need a logic system behind the behaviour of the animals of each species. A simple example of how this could be accomplished will be presented.

In a forest biome, two species of animals are added, wolves and rabbits. To each animal we give four variables, three which take a float value between 0 and 1, one for hunger, other for thirst and one for reproductive predisposition, 0 being the lowest and 1 the highest. Lastly, the rabbits have an integer variable for their age. All float values decrease while the rabbit moves around on the terrain. If either hunger or thirst reach 0 the animal dies and is erased from the terrain, if he manages to stay alive for a number of counted frames, the age is incremented by one, when the age reaches a certain number the rabbits increase their size and they are flagged as adults. The rabbits are randomly instanced on the land with all these variables at 0.5, and a path-finding algorithm is implemented so the rabbits move on the terrain to a water body or to a plant, so they can improve their stats. Every time thirst and hunger are at 1, the reproductive stat increases and when it also reaches 1, if the rabbit is flagged "adult", the rabbit will search for another rabbit and instance a new baby rabbit. Wolves do not eat plants, so their behaviour would be similar to the rabbits, but instead, every time a wolf got close to a rabbit, there would be a chance for the wolf to consume the rabbit, increase its hunger value and the rabbit would die.

This is a simple yet effective way of controlling an ecosystem, as this same logic can be applied to any set of two or more animals in any biome, like the ocean or a desert, we just need a lower level animal in the food chain and some higher one to control the lower's population and to reproduce themselves. After some tweaking so none of the species overpopulates one area, a

self controlled simple ecosystem is achieved, which can be flexible to include other species.

Unrelated to the fauna present in the terrain, it would add up to the ecosystem if a deep study about each biome's native species of plants was made, in order to include all of the predominant species related to each region, adding more variation and a better degree of realism, since we currently have only one species of the most predominant tree in each biome.

6.2.2 Implementation of the Marching Cubes Algorithm

The Marching Cubes algorithm [22] presented by Lorensen and Cline in 1987, is used for extracting a polygonal mesh of an isosurface from a three-dimensional grid, each element of this grid being a voxel. The space is divided in a grid, but instead of instancing a block in each parcel, to create the terrain, it uses each space to build a piece of the terrain, and in the end, all the pieces together form the final landscape.

The use of this algorithm would be very beneficial, because it would add realism to this work, since the terrain would have a more accurate look without losing any of the capabilities the sketching phase provides, of ordering the terrain with different biomes, since it is still made of voxels but not full cubic blocks. If the intention of the user is to actually have a world/terrain made of blocks, because it better fits his needs, this would not be a very good change to this framework. With this in mind, the addition of the Marching Cubes algorithm could be done in parallel with the voxel block approach, giving the user the choice of generating a surface with the method he wants, while still using the user sketch feature.

In Figure 6.1 we can see a terrain example of the video game Valheim, achieved using a voxel-based approach implementing the Marching Cubes algorithm. Although not the most realistic a terrain generator can be, it produces much better results on those regards compared to our cubic voxel approach.



Figure 6.1: Procedurally Generated landscape using a voxel-based method, implementing the Marching Cubes algorithm. Image from <https://www.corrosionhour.com/valheim-a-survival-success-story/>.

6.2.3 Implementation of Weather and a Night and Day System

Exactly as the addition of clouds adds to a terrain generator, even though the clouds are not actually part of the landmass itself, the inclusion of a basic night and day cycle and a simple weather system can massively improve the immersion and overall feel of an environment, when experiencing it from a "player" perspective, like one of the views detailed in Section 4.7 describes.

To implement the night and day cycle, it could be done similarly to how the real world operates, as in having a light source, a large spot light acting as the "Sun", that rotates around the terrain lighting every spot, and dimming the light as it crosses the horizon and sets. This is a very rough and basic way of accomplishing it, but it certainly adds to the overall result.

Changing the weather could be done by adding a particle effect to simulate rain dropping from the sky and change the colour of the sky box from blue to gray as well as add more layers of clouds. The weather could turn from sunny to rainy with the help of a flag that has the chance to become true with a random number generator function, every time the sun rises. The rain particle effect would be changed to snow falling in very cold biomes, like the Taiga or the Alpine Tundra, or could even change when the player crosses a certain height value, on top of mountain regions.

6.2.4 Implementation of the City "Biome"

Having the option to use a new colour for the sketch that translates to the generation of a city would be a very convenient addition to this framework. However, including a city is not as straightforward as including any natural biome, since, to start, a city needs some form of organization on its base, so it can be coherent and comparable to real life city standards. An urban landscape also can not be generated using only noise methods to control the terrain elevation and features, since we need to generate actual buildings and place them in our previously ordered terrain, in a meaningful fashion, so we can have at least a basic rudimentary organization in the city, and not random buildings being spawned in an area.

The difficulty of adding a city area starts with the sketch phase, because if an user can colour any area he wants, interpreting the coloured area and organizing it, so it can have streets and buildings, is highly subjective. Imagine if the user decides to paint a small area, as an extreme example, we will say the user only paints one pixel with the colour that translates to a city area. It does not make sense to generate a city on one block, so the sketch phase would need to have a restriction regarding the size of the painted area. Also, in the sketch phase, the shape of the area painted by the user to be a city, would also need some form of restriction. Again, using an extreme example, let us imagine if the user painted a straight line of one pixel on the canvas. If it was any other biome, it would be completely fine (it would not be realistic, and as was mentioned, the sketch "realism" is a task for the user, but it would be completely possible),

but a city can not be ordered properly if everything we have to work with is an area of a line. So to start, we would need strong restrictions for the shape and size of a city area.

Given an area for the city, we would first need a way to procedurally generate roads, with some form of coherence, using methods and algorithms similar to [18]. Next, depending on the approach, we would need a way to generate the buildings, and here, we have some options. We can either instance previously built 3D meshes of different buildings inside each lot, between roads, resulting on a not so varied city, with a lot of the same buildings repeating (also, depending on the quantity of available buildings), or we could procedurally generate different structures, like the ones in Figure 6.2 using L-Systems consisting of modules for transformation, extrusions, branching and terminations, and finally, modules for templated structures like antennae and roofs as was done firstly in [32].

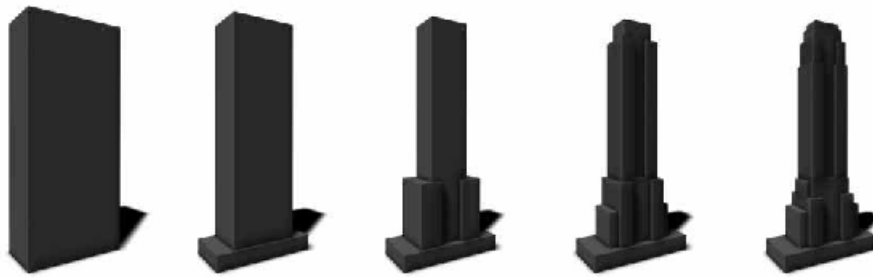


Figure 6.2: Procedurally generated buildings using L-Systems. Image from [32].

6.2.5 Optimization of Execution Times

Perhaps the most straightforward improvement that could be made, since this framework does not have the best execution times for larger terrains, as was seen in Table 5.1 and was not built to deliver good performance, and more as proof of concept.

A good optimization that could be done would be to implement the code using parallel computation notions, dividing the work across different threads, so it can be computed more efficiently, especially the part of the logic where the surface positions are compared to the positions inside every list, since that is the part of the execution which takes more time to complete.

Another advantageous optimization to the framework, a feature which would be mandatory, if it was used in a project like a game, would be to only spawn the chunks that are close to our player, and delete the ones that are farther away. This optimization would not make much sense to implement in this phase, since we are assessing terrains as a whole to understand its characteristics and how the user can influence them. Implementing it would not give the possibility to look at the complete terrain, all the landscape at the same time

6.3 Final Thoughts

Procedural Generation is a vast and dense area, finding its use on more and more games and applications as time progresses, since the need for quickly generated, detailed and good quality assets is rising, now more than ever. As worlds and detail in games tend to get bigger, it becomes mandatory to have good systems to generate portions of work that would translate to huge monetary costs especially in staff. Obviously, terrain generation using procedural methods is no exception and it is interesting to understand how the advance in technology will affect this area, since the level of detail landmasses that were generated procedurally have now, compared to before, is astonishing. More specifically, it arouses more curiosity how the generation of terrains, using input from users will evolve, hopefully reaching a day where a person with no knowledge in technology whatsoever, can, with the help of a very simple interface, generate a full blown and detailed landmass, where he decided every single feature of that terrain, as well as topography, vegetation, fauna, weather and everything one can imagine. It could have huge impact on the military for example, in simulations for explanations or training. Only the future will tell how it is going to evolve, but it most definitely looks promising.

Bibliography

- [1] Oscar Argudo, Carlos Andujar, Antonio Chica, Eric Guérin, Julie Digne, Adrien Peytavie, and Eric Galin. Coherent multi-layer landscape synthesis. *The Visual Computer*, 33(6): 1005–1015, 2017.
- [2] Michael Becher, Michael Krone, Guido Reina, and Thomas Ertl. Feature-based volumetric terrain generation and decoration. *IEEE transactions on visualization and computer graphics*, 25(2):1283–1296, 2017.
- [3] Fares Belhadj and Pierre Audibert. Modeling landscapes with ridges and rivers. In *Proceedings of the ACM symposium on Virtual reality software and technology*, pages 151–154, 2005.
- [4] Kazimierz Choros and Jacek Topolski. Parameterized and dynamic generation of an infinite virtual terrain with various biomes using extended voronoi diagram. *J. UCS*, 22(6):836–855, 2016.
- [5] Guillaume Cordonnier, Jean Braun, Marie-Paule Cani, Bedrich Benes, Eric Galin, Adrien Peytavie, and Eric Guérin. Large scale terrain generation from tectonic uplift and fluvial erosion. In *Computer Graphics Forum*, volume 35, pages 165–175. Wiley Online Library, 2016.
- [6] Guillaume Cordonnier, Eric Galin, James Gain, Bedrich Benes, Eric Guérin, Adrien Peytavie, and Marie-Paule Cani. Authoring landscapes by combining ecosystem and terrain erosion simulation. *ACM Transactions on Graphics (TOG)*, 36(4):1–12, 2017.
- [7] Oliver Deussen, Pat Hanrahan, Bernd Lintermann, Radomír Měch, Matt Pharr, and Przemyslaw Prusinkiewicz. Realistic modeling and rendering of plant ecosystems. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 275–286, 1998.
- [8] Rahul Dey, Jason G Doig, and Christos Gatzidis. Procedural feature generation for volumetric terrains using voxel grammars. *Entertainment Computing*, 27:128–136, 2018.
- [9] Jonathon Doran and Ian Parberry. Controlled procedural terrain generation using software agents. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(2):111–119, 2010.

- [10] Roland Fischer, Philipp Dittmann, René Weller, and Gabriel Zachmann. Autobiomes: procedural generation of multi-biome landscapes. *The Visual Computer*, 36(10):2263–2272, 2020.
- [11] Alain Fournier, Don Fussell, and Loren Carpenter. Computer rendering of stochastic models. *Communications of the ACM*, 25(6):371–384, 1982.
- [12] Jean-David Génevaux, Eric Galin, Eric Guérin, Adrien Peytavie, and Bedrich Benes. Terrain generation using procedural models based on hydrology. *ACM Transactions on Graphics (TOG)*, 32(4):1–13, 2013.
- [13] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014.
- [14] Saskia Groenewegen, Ruben Michaël Smelik, Klaas Jan de Kraker, and Rafael Bidarra. Procedural city layout generation based on urban land use models. In *Eurographics (Short Papers)*, pages 45–48, 2009.
- [15] Stefan Gustavson. Simplex noise demystified. *Linköping University, Linköping, Sweden, Research Report*, 2005.
- [16] Mark Hammel, Przemyslaw Prusinkiewicz, William Remphrey, and Campbell Davidson. Simulating the development of *fraxinus pennsylvanica* shoots using l-systems. In *Proceedings of the sixth western computer graphics symposium*, pages 49–58, 1995.
- [17] Alex D Kelley, Michael C Malin, and Gregory M Nielson. Terrain simulation using a model of stream erosion. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 263–268, 1988.
- [18] George Kelly and H McCABE. An interactive system for procedural city generation. *Institute of Technology Blanchardstown*, 25, 2008.
- [19] Lin Ziwen Kelvin and Bhojan Anand. Procedural generation of roads with conditional generative adversarial networks. In *2020 IEEE Sixth International Conference on Multimedia Big Data (BigMM)*, pages 277–281. IEEE, 2020.
- [20] Christopher G Langton, Charles Taylor, Doyne Farmer, and Steen Rasmussen. *Artificial life II*. Addison-Wesley Longman Publishing Co., Inc., 1991.
- [21] Aristid Lindenmayer. Mathematical models for cellular interactions in development i. filaments with one-sided inputs. *Journal of theoretical biology*, 18(3):280–299, 1968.
- [22] William E Lorensen and Harvey E Cline. Marching cubes: A high resolution 3d surface construction algorithm. *ACM siggraph computer graphics*, 21(4):163–169, 1987.
- [23] B.B. Mandelbrot, W.H. Freeman, and Company. *The Fractal Geometry of Nature*. Einaudi paperbacks. Henry Holt and Company, 1983. ISBN: 9780716711865.

- [24] Benoit B Mandelbrot and Benoit B Mandelbrot. *The fractal geometry of nature*, volume 1. WH freeman New York, 1982.
- [25] Alexandre Philippe Mangra, Adrian Sabou, and Dorian Gorgan. Tsch algorithm-terrain synthesis from crude heightmaps. *Romanian Journal of Human-Computer Interaction*, 9 (2), 2016.
- [26] Ivo Marák, Bedřich Beneš, and Pavel Slavík. Terrain erosion model based on rewriting of matrices. 1997.
- [27] Radomír Měch and Przemysław Prusinkiewicz. Visual models of plants interacting with their environment. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 397–410, 1996.
- [28] F Kenton Musgrave, Craig E Kolb, and Robert S Mace. The synthesis and rendering of eroded fractal terrains. *ACM Siggraph Computer Graphics*, 23(3):41–50, 1989.
- [29] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. *ACM Trans. Graph.*, 25:614–623, 07 2006. doi:10.1145/1141911.1141931.
- [30] Jacob Olsen. Realtime procedural terrain generation. 2004.
- [31] Emmanouil Panagiotou and Eleni Charou. Procedural 3d terrain generation using generative adversarial networks. *arXiv preprint arXiv:2010.06411*, 2020.
- [32] Yoav IH Parish and Pascal Müller. Procedural modeling of cities. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 301–308, 2001.
- [33] Ken Perlin. An image synthesizer. *ACM Siggraph Computer Graphics*, 19(3):287–296, 1985.
- [34] Ken Perlin. Noise hardware. *Real-Time Shading SIGGRAPH Course Notes*, 2001.
- [35] Adrien Peytavie, Eric Galin, Jérôme Grosjean, and Stéphane Mérillou. Arches: a framework for modeling complex terrains. In *Computer Graphics Forum*, volume 28, pages 457–467. Wiley Online Library, 2009.
- [36] Przemysław Prusinkiewicz and Mark Hammel. A fractal model of mountains and rivers. In *Graphics Interface*, volume 93, pages 174–180. Canadian Information Processing Society, 1993.
- [37] Przemysław Prusinkiewicz and Aristid Lindenmayer. *The algorithmic beauty of plants*. Springer Science & Business Media, 2012.
- [38] Przemysław Prusinkiewicz, Mark S Hammel, and Eric Mjolsness. Animation of plant development. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 351–360, 1993.

- [39] Kristine Faith J Roque and Rowel O Atienza. Real-time procedural generation of a growing corn model. In *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 1–2, 2017.
- [40] Pascale Roudier, Bernard Peroche, and M Perrin. Landscapes synthesis achieved through erosion and deposition process simulation. In *Computer Graphics Forum*, volume 12, pages 375–383. Wiley Online Library, 1993.
- [41] Stuart Russell and Peter Norvig. Artificial intelligence: a modern approach. 2002.
- [42] Ruben M Smelik, Klaas Jan De Kraker, Tim Tutenel, Rafael Bidarra, and Saskia A Groenewegen. A survey of procedural methods for terrain modelling. In *Proceedings of the CASA Workshop on 3D Advanced Media In Gaming And Simulation (3AMIGAS)*, volume 2009, pages 25–34, 2009.
- [43] Alvy Ray Smith. Plants, fractals, and formal languages. *ACM SIGGRAPH Computer Graphics*, 18(3):1–10, 1984.
- [44] Ondřej Št'ava, Bedřich Beneš, Matthew Brisbin, and Jaroslav Křivánek. Interactive terrain modeling using hydraulic erosion. In *Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 201–210, 2008.
- [45] Georges Voronoi. Nouvelles applications des paramètres continus à la théorie des formes quadratiques. premier mémoire. sur quelques propriétés des formes quadratiques positives parfaites. *Journal für die reine und angewandte Mathematik (Crelles Journal)*, 1908(133):97–102, 1908.
- [46] Paul Walsh and Prasad Gade. Terrain generation using an interactive genetic algorithm. In *IEEE Congress on Evolutionary Computation*, pages 1–7. IEEE, 2010.
- [47] Hua Fei Yin and Chang Wen Zheng. A practical terrain generation method using sketch map and simple parameters. *IEICE TRANSACTIONS on Information and Systems*, 96(8): 1836–1844, 2013.
- [48] Howard Zhou, Jie Sun, Greg Turk, and James M Rehg. Terrain synthesis from digital elevation models. *IEEE transactions on visualization and computer graphics*, 13(4):834–848, 2007.