Technical University of Crete
School of Electrical and Computer Engineering

Procedural Generation Algorithms in Games and 3D Graphics

Diploma Thesis by
Ioannis Markou

Chania, Crete, October 2023

Abstract

In this thesis, a Procedural Environment Generator is created to be used as a tool by game developers who want to automatically generate different types of terrain with the appropriate vegetation. Nowadays, games are becoming more and more complex both in gameplay and development, thus creating the demand for tools to help developers streamline and speed up various parts of the development process. This tool focuses on terrain generation and the population of said terrain with vegetation. Regarding the terrain, features to control its size, altitude difference between highest and lowest points, mountain and hill height and detailing have been implemented. Additionally, the map that gets created can be split into different areas. Each area has its own material and zones of vegetation, meaning that large environments with different sub-environments can be generated (e.g going from a jungle environment to a desert or snowy environment). Vegetation zones can be created in each one of the areas, meaning that each sub-environment can have different vegetation withing specific height limits. Furthermore, the user can add their own 3D models and prefabs of both trees and vegetation, such as grass, flowers, mushrooms or anything that fits their custom environment.

Acknowledgements

For this thesis, first and foremost, I would like to thank Prof. Katerina Mania for giving me the space and time to develop my idea in a field that is just beginning to rise and also for her invaluable support. Furthermore, I need to also thank Prof. Michail Lagoudakis and Assistant Prof. Nikos Giatrakos for accepting to be on my committee.

I would also like to thank my colleagues in the Surreal research team for their guidance and especially Minas Katsiokalis, Yannis Kritikos and Andreas Polychronakis.

Finally, I would like to thank my friends and especially my family, my parents Rena and Pavlos, and my brother Manos for their endless support throughout my studies and the writing of this thesis, both of which would be impossible without them.

Declaration of Authorship

I, Ioannis Markou, declare that this thesis titled, "Procedural Generation Algorithms in Games and 3D Graphics" and the work presented in it are my own. I confirm that:

- This work was done wholly while in candidature for a research degree at this University.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.

Jury

3-member Committee

Professor Katerina Mania (Thesis Supervisor)	ECE, Technical University of Crete
Professor Michail G. Lagoudakis	ECE, Technical University of Crete
Assistant Professor Nikos Giatrakos	ECE, Technical University of Crete

Contents

List of Figures

Chapter 1

Introduction

This thesis focuses on creating a Procedural Content Generation(PCG) tool, specifically to generate custom environments, using the Unity Game Engine. This tool is to be used by developers inside Unity3D to create natural environments that fit their purpose. That includes generating a terrain, dividing it into different areas and populating it with vegetation.

1.1 Brief Description

This section will provide a brief description of the tool developed for this thesis. First, let's begin by talking about the terrain. The terrain consists of multiple meshes, called blocks, being combined. The generated terrain's size is $N \times N$ blocks. Additionally, each block has $M \times M$ vertices. Both M and N are set by the user. These vertices work as points to spawn vegetation on. However, only doing these will result in a large flat surface, and here comes the challenge of procedurally adding detailing: hills, valleys, mountains and everything in between. That's where noise comes in.

Most Procedural Terrain Generation (PTG) algorithms/tools make use of some kind of noise algorithms. Natural terrains consist of irregular shapes and have a fractal nature, making it difficult for artists and developers to manually create them digitally. By using gradient noise functions, we can use the luminosity of each pixel in the output to generate a heightmap[2] [3]. While there are many different noise algorithms to use, after testing a couple, this thesis makes use of the Perlin Noise algorithm which will be explained in detail later.

So after generating the terrain, giving it detail with noise, it's time to populate it with vegetation. Firstly, the user can split the generated terrain into different areas. Each area has

its own borders, material and zones of vegetation based on height. On each vegetation zone, the probability of spawning vegetation, 3D models and height limits are customizable. The algorithm goes through each vertex of each block and depending on the probability given either spawns a batch of vegetation or not. Finally, the option to use a specific seed or use a random one is available.

Now let's get more into the specifics and the user experience. After importing the tool into Unity, the user will be able to launch a Custom Window that shows all the available options for generating an environment. Once the values are set, the generated environment will appear once Unity's Play button is pressed. In order for the user to make any kind of changes to the environment, they need to stop Unity and make the changes inside the Custom Window offline, then re-run Unity. This thesis can be summed up in the figures 1.1 and 1.2 shown below. To sum it all up, we have created a custom window that works as an interface between the user and the program. The program initially generates the mesh, based on the user's inputs (Blocks and Vertices Per Block). Following that, the mesh is shaped based on Noise Density, Mountain Intensity and Hill Detail making it rougher, smoother, taller or shorter. Finally, the user must create empty game objects to attach Area or Vegetation script in order to divide the terrain into sub-environments and each sub-environment to zones of vegetation based on height.

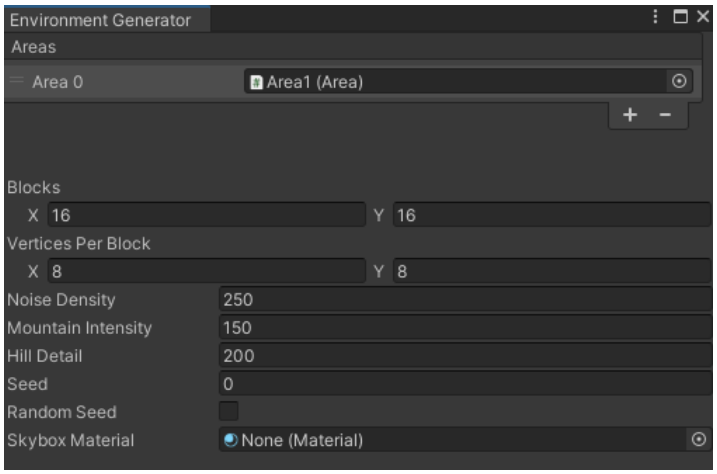

Figure 1.1: Example of the Environment Generator window values.

Figure 1.2: Example of a generated environment.

1.2 Structure of the Thesis

In this section of the thesis, an overall structure of the document will be provided with a brief description.

In Chapter 2, we provide the research behind this thesis and introduce the reader to the field of Procedural Content Generation (PCG), its history, purpose and future goals. Additionally, we discuss what Noise is, its use on the field and analyze two famous noise algorithms: Perlin Noise and Diamond Square. Finally, we show how Minecraft generates its terrain, which is closely related to the work done in this thesis.

In Chapter 3, initially we present the case analysis followed by a thorough explanation of how to use the tool developed in this thesis. Additionally, each field of the custom editor window that acts as an interface between the user and the program is analyzed, showing how different values on each field can yield very different environment outputs.

In Chapter 4, we provide a detailed description of the structure of the system. More specifically, we show how the main components communicate with each other and dive deep in explaining how each of them works and what their purpose is.

In Chapter 5, we implement two demos, two different environments to show some of the capabilities of this tool. The first demo presents a scenario where there's a fire spreading across a green, healthy environment. The second demo presents a snowy environment, slowly transitioning into a greener environment, showing off many different areas.

Finally, in Chapter 6, an overall conclusion of the thesis is given. We also provide ideas of how this tool can be improved in the future and discuss its limitations.

Chapter 2

Research Overview

The research of this thesis mainly focuses on Procedural Content Generation (PCG) algorithms. This chapter will provide a history and explain the use of the field. Additionally, a dive into different noise functions and their uses will be discussed.

2.1 Procedural Content Generation

Defining Procedural Content Generation is a rather difficult task, since it contains many loose definitions itself such as "content". The best thing we can do is to limit the borders in which PCG lies. To begin with, let's talk about human input and PCG. There are many algorithms and tools that require some human input to procedurally generate content, meaning that the result of the algorithm should not be completely automatic in order to be considered "PCG". This is called mixed-initiative PCG[8]. However tools with complete control given to the user such as map or terrain editors, are not considered PCG. In this thesis, we make use of mixed-initiative PCG as the user defines size, shaping and vegetation parameters after allowing the algorithm to generate the environment.

Moving on to another factor of PCG, let's discuss about randomness. PCG is not about making random decisions just for the sake of replayability or giving away the control to an algorithm. Instead such generators use pseudo-random tools to create content following strict constraints[8]. This is called stochasticity. In this thesis, we use Unity's Random library to help decide whether or not vegetation spawns but what kind of vegetation spawns or the probability of spawning is decided by the user.

Taking into account the matters discussed above, we can attempt to give a definition of PCG. Procedural Content Generation refers to game content being generated using computer procedures, as the name suggests. More specifically, it's all about a program, software or algorithm generating stochastic game content with limited or no user input[1]. However, the term "content" is not strictly defined. In general it includes game parts such as terrain, music, animations even textures or whole stories but it excludes parts like Non-Playable Characters (NPC) AI behavior.

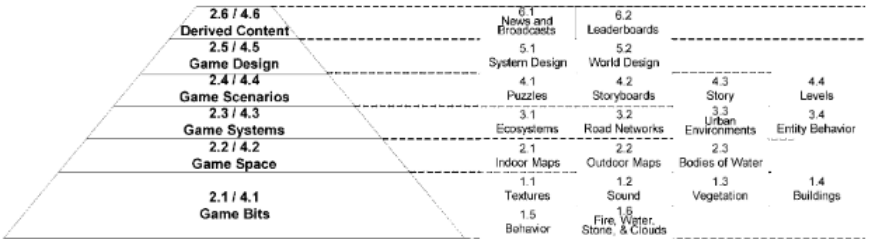

Figure 2.1: Possible procedural generated game content as shown by Hendrikx, Meijer, van der Velden, and Iosup [5].

In this thesis, we have created a PCG tool that can be used by anyone inside Unity in order to generate custom rural environments. Both the terrain and the vegetation are procedurally generated according to the settings set by the user.

2.1.1 History

Originally, PCG came up as a solution to technological bottlenecks in the 1980s. More specifically, computer hardware at the time limited the creation of bigger games both in graphics and content. In Elite (Acornsoft 1984), a space fighting game, instead of storing data such as planet positions or names, developers came up with the idea of using a PCG algorithm that generates all the necessary data given a specific seed[1]. Another game infamous for using PCG algorithms is named Rogue (Pixel Games UK 1985). In Rogue, levels consisting of a number of rooms connected via hallways in a 2D environment that are called Dungeons are procedurally generated. We can see example of those two games in the figures 2.2, 2.3.

In modern era, many commercial games use more sophisticated PCG algorithms in sound, map generation, item placement e.t.c. There are even games, like No Man's Sky that base the entire experience on PCG. No Man's Sky by Hello Games features massive worlds, entire planets with their own vegetation and wild life for the player to explore and survive. Every possible

ecosystem, galaxy, structure, biome and more is procedurally generated and the developers claim that there are 18 quintillion possible planets, rendering the game endless.

Another game that uses PCG to generate game scenarios is Left 4 Dead(Valve 2008). Left 4 Dead is a 4 player co-op FPS video game, that places 4 players against hordes of zombies trying to survive the zombie apocalypse. When playing Left 4 Dead, no run is the same. Each time, weapons, health items, ammo e.t.c are placed in different spots, zombie spawns both in quantity and place are never the same, and possibly parts of maps could be cut off. This is done by an AI system called The Director. The Director measures the Survivor Intensity,stress level and skill of each player, and decides on whether to make the game easier or harder. This PCG ensures great replayability and immersion in the world of Left 4 Dead[15]. There's even research been done to combine biological indicators (with the appropriate hardware) and The Director, meaning actual biological factors such as facial expressions, eye movement, heart rate e.t.c are used as data for the Director to make decisions[16].

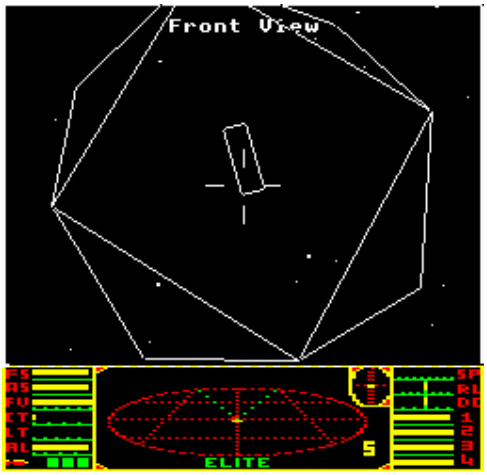

Figure 2.2: The BBC Micro version of Elite.

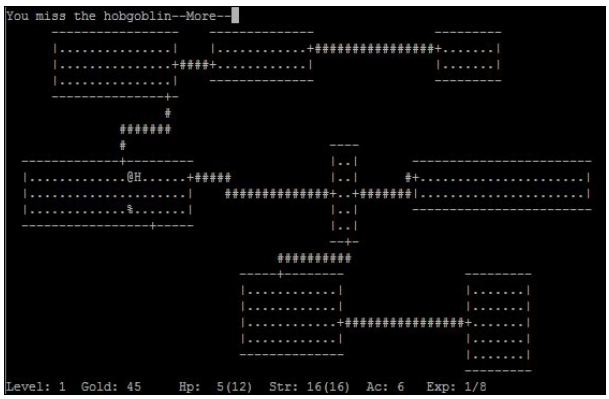

Figure 2.3: Procedurally generated dungeon in Rogue.

2.1.2 Why use Procedural Content Generation?

The explanation of PCG gives birth to the question: why use it? While there may be some obvious reasons, PCG can be useful in many different situations. Let's dive deeper.

To begin with, the gaming and 3D graphics industry has seen immense growth in popularity over the last couple of decades[9]. Game development companies nowadays, consist of hundreds

Figure 2.4: No Man's Sky.

Figure 2.5: Minecraft.

of people with different fields of expertise that work together for, possibly, years to create a singular game. For example, a popular AAA modern game called Call of Duty has multiple different companies and 2-3 years to develop a singular sequel. With the advancement of computer hardware, games also become more complex both in gameplay and graphics, requiring even more people to work on them. This rise in popularity and complexity has given birth to problems regarding scheduling, management, quality and budget in video game companies[10]. These issues led to a need for software tools to aid in the design and development of games. Nowadays, games are rarely built from the bottom up and tools like game engines abstract common game features like rendering or input handling, allowing the developers to focus their time elsewhere[11]. Regarding PCG, sophisticated PCG tools can help overcome the bottlenecks of budget, manual labor and time by automating procedures otherwise handled by humans[5].

By having the algorithms make decisions about various game parts on the run, instead of developers pre-determining things like spawn points, when or where music plays, routes e.t.c, PCG can greatly increase replayability by making each run of a game unique. Furthermore, by combining PCG with neural networks or machine learning and taking into account various player's factors such as stress level, skill, enjoyment e.t.c, we can greatly customize the content that gets procedurally generated to fit their own playstyle[1].

2.1.3 Procedural Terrain Generation

One of the most basic elements of games is the ground that you walk, run and stand on. Terrain generation requires lots of work and if an infinite terrain is desired, then it is impossible with manual labor. That's where PCG algorithms can help with this tiresome process. If we imagine

a terrain as a two-dimensional grid, we can figure out ways to fill in the cells with values, where each value would correspond to a specific height. Positive values would be mountains and hills, negative values valleys, and numbers close to zero would represent flatter surfaces.

But how do we fill those cells with values? The first idea that naturally comes to mind is to fill the cells with random numbers. While that would, technically, create a terrain it wouldn't be visually or practically useful. The terrain would consist of random highs and lows, like random mountain spikes and flat points with no connection to each other, something that does not occur in natural terrain.To create more natural looking environments, Procedural Terrain Generation (PTG) usually uses some kind of gradient noise function to generate a grayscale heightmap and use that data to map a terrain[2]. There are many different noise algorithms like Perlin Noise, Diamond Square, Simplex Noise e.t.c that have been developed over the years.

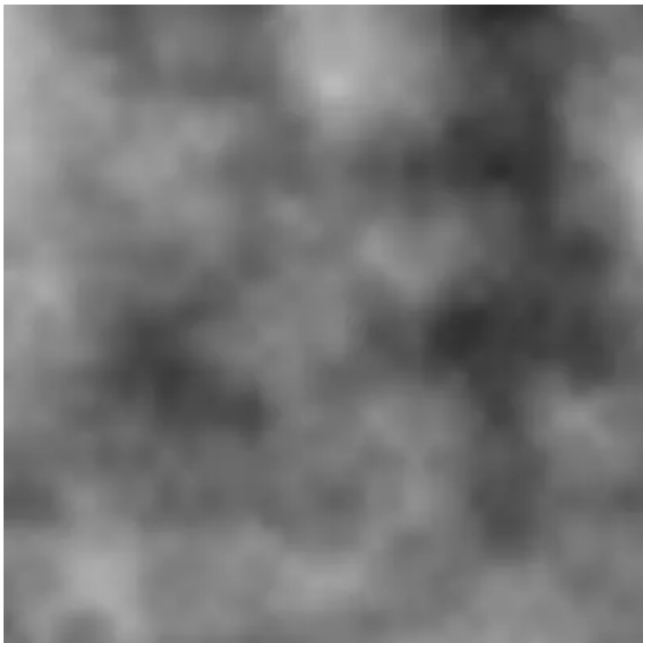

Figure 2.6: Perlin Noise, grayscale.

Generated terrains should also contain both stochasticity and determinism to a certain degree. To be more specific, every time a terrain gets generated by a potential terrain generator, it should feel new to the user (given that a random seed is used). However, complete randomness would lead to unplayable environments. This means that the generator should adhere to internal rules set by the developer in order to maintain a specific structure. Finally, tools that generate

terrain should have options that are easy to use and can be adjusted by the user to create the terrain that fits their needs[12].

In this thesis we initially generate a mesh whose size and vertices are set by the user. All of the setting that the user can mess around with are in a custom, simple editor window we have implemented with straight-forward options that can create random terrains with specific looks.

2.1.4 Future goals, challenges and conclusions

PCG in the present is mostly only a feature in modern games, aiming to make them more challenging and add a sense of discovery and fellowship while players find out all the possible environments, item placements, game scenarios e.t.c that a PCG feature can offer[4], [14].

PCG in the future offers exciting scenarios and possibly a new game genre dedicated to it. To be more specific, we can imagine procedurally generating not only content, but whole games, different games with different feels every time a button is pressed. Maybe even generating game engines themselves, rules and ideas[6].

In the more foreseeable future, however, can help bridge the gap between players thirsting for new game content and the huge manual labour that developers have to undergo in order to keep up with those demands. By creating tools like the one proposed in this thesis, fine-tuning them and giving as much control as possible to the designer in order for them to be able to generate the content they would otherwise generate manually. PCG can also be used in serious games to generate personalized scenarios for teaching and educational purposes[13]

2.2 Noise

Noise in games and graphics is a common technology used to add detailing in various 3D surfaces like skyboxes and terrains or add irregularities in otherwise perfect 3D models thus making them more realistic[3]. Ken Perlin with his infamous Perlin Noise has helped a lot with procedural texturing, and a very famous example is his marble vase which was textured using Perlin noise (2.7). By the term procedural, when it comes to noise, we mean that the values of each pixel are determined by using mathematics and algorithms[17].

Noise as a solution to procedurally generated terrains came up due to the fractal nature of natural terrains. By the term fractal we mean shapes with similarities in various scales. To be more specific, let's imagine a mountain that if seen from afar has peaks and valleys. However, if we hike through these peaks and valleys, we will meet hills and ravines. And that is what

fractal means: self similarity in different scales[2]. By using noise algorithms, we can generate pseudo random fractal textures. Two noise algorithms were tested to procedurally generated terrain: Perlin Noise and Diamond Square.

Figure 2.7: Marble vase procedurally textured with Perlin Noise.

2.2.1 Diamond Square Algorithm

Let's start by explaining the "most fractal" algorithm, Diamond Square. We start with an empty grid that has a size that satisfies the formula $2^n + 1$ where n = 1,2,3... due to the way that the algorithm breaks down the heightmap to perform calculations. Then we assign random values to the four corners of the grid and proceed by performing the diamond step. As you can see in the image below, this creates a big square on the grid. Then, the diamond step fills the value of the square centre with the average of the values of each corner. Finally, with the square step we fill the missing corner points by averaging the centre point with the neighbouring points of the corner we wish to fill. These steps repeat recursively until the whole grid is filled with values. In the figure 2.8 we can see the steps more clearly.

However, the tool created in this thesis does not use this algorithm because after testing it, it ended up creating too rocky and harsh environments that could possibly be useful for Flight Simulators where the player only has to see the terrain from a distance and not interact with it.

2.2.2 Perlin Noise Algorithm

One of the most famous and classic algorithms for noise generation is Perlin Noise. Perlin Noise was first done by Ken Perlin (thus the name) and it is a gradient noise generator, meaning the algorithm first generates gradient vectors and then calculates the value of the pixel, that produces fractal results. Perlin Noise is a popular choice among developers, and after testing it and seeing its flexibility and how you can create any type of terrain you wish with it, I chose it for this tool.To explain it, let's consider the grid of pixels we want to fill with values. The algorithm first divides the grid into larger rectangles as shown in the figure 2.9. The frequency of the rectangles directly impacts the noise output. The higher the frequency, the more detailed the final output will be due to the creation of more pseudorandom gradient vectors. Pseudorandom gradient vectors are generated in the purple corners that are shown on the image above and they define a positive direction. The algorithm then continues by creating distance vectors, which are vectors from the point/pixel we want to calculate the value towards the four corners of the rectangle. Finally, the dot product between the gradient vector and its corresponding distance vector is calculated for every corner and we interpolate between the four values to get a weighted average (figure 2.10).

In this thesis, the Diamond Square algorithm was tested but deemed unfit for our vision so we used the Perlin Noise algorithm. Through the custom editor window, the user can affect the shaping of the terrain by adjusting various parameters of the noise generation.

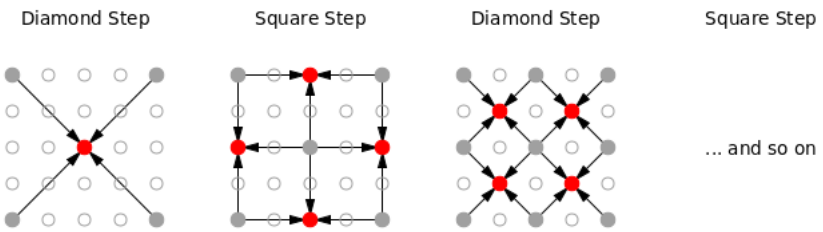

Figure 2.8: Diamond square algorithm steps.

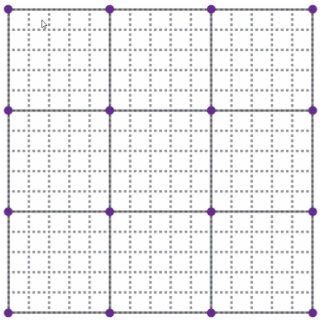

Figure 2.9: Original grid divided into rectangles.

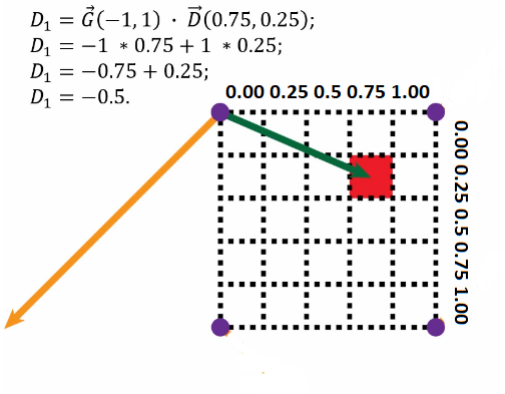

Figure 2.10: Gradient vectors : orange, Distance vector: Green.

2.3 Minecraft

A great inspiration for this thesis was Minecraft by Mojang Studios. Minecraft both manages to create terrain using Perlin Noise and populate said terrain with biomes, vegetation and wildlife.

Using Henrik Kniberg's video, game developer and designer at Mojang studios, let's give a short explanation to Minecraft's terrain generation. A Minecarft world is about 3.6 billion square killometers and there are 18 quintillion different worlds that can be generated. The world is generated in big chunks, as he calls them, 16x16 wide and 384 blocks high. A block can be seen in the figure 2.11.

Figure 2.11: Minecraft chunk.

The first step is terrain shaping. In order for the terrain to have the realistic randomness of nature but also a certain smoothness, Perlin Noise is used. By taking a look at a 1D noise field example in figure 2.12, we can clearly see that the data provided by Perlin Noise could represent hills, mountains and valleys in a 2D game. In the same way, 2D noise fields can create heightmaps for 3D environments. To add further detailing, octaves are used. Let's assume a simple sin that represents terrain height. An octave is another curve with higher frequency and less amplitude. Both sins are a sequence of valleys and mountains.

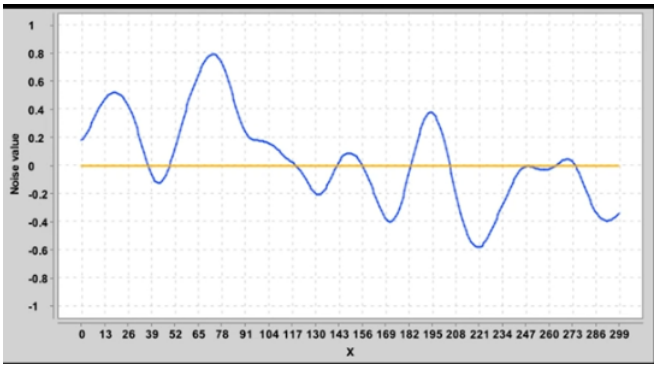

Figure 2.12: 1D noise field.

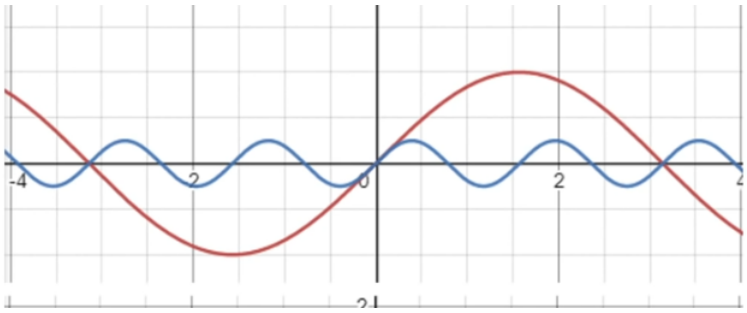

Figure 2.13: Two sins representing terrain data.

However, by adding these two curves we get a result with much more detailing as we can see in figure 2.14. As we can imagine, adding more and more octaves (up to a point) can help have mountains, hills and more with great details.

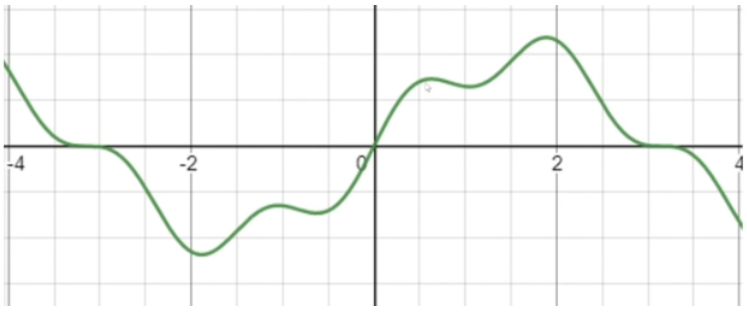

Figure 2.14: Result after adding the two curves.

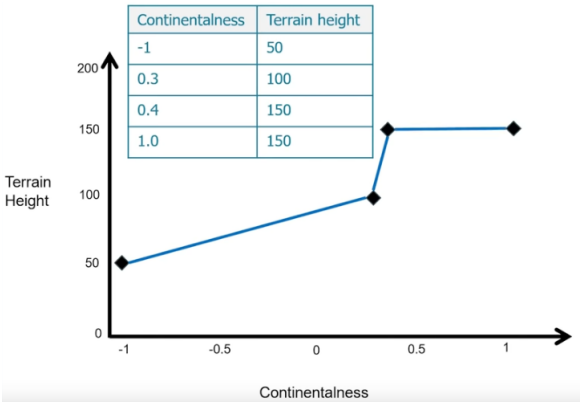

Figure 2.15: Continentalness.

To make more interesting terrain, developers used spline points. To be more specific, the developer enters points in specific locations and a curve is created that follows these points. In general, 2D Perlin Noise functions take x and y coordinates as an input and give an output between 1 and -1. Naming this output "Continentalness" , Minecraft developers created points on a grid where x is the terrain height and y the continentalness. Finally, they used the spline curve mentioned before to connect these points, thus creating bigger variations in the produced terrain by adding steeper changes as can be seen in the figure 2.15.

To add even more variation, Minecraft uses 3 different noise maps (figure 2.16) with different octaves in combination with spline curves: continentalness, erosion and PV (peaks and valleys).

In this thesis, we have used the same logic as Minecraft's in terrain generation and shaping. All images are taken from Henrik Kniber's video: Minecraft terrain generation in a nutshell.

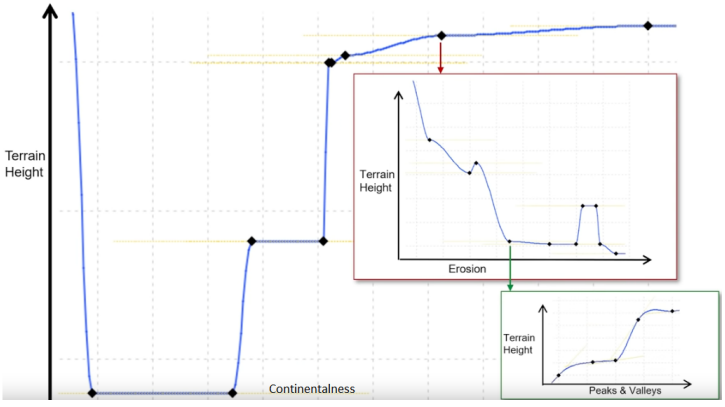

Figure 2.16: Final result.

Chapter 3

Use Case

3.1 Introduction

After talking about the field of PCG and the influence behind this thesis, this chapter will provide further information on how this tools is supposed to be used.

In general, this procedural environment generation tool developed in this thesis is targeted towards individuals that want to have a custom 3D environment without going into the trouble of making it themselves. The tool provides a blank canvas to work with, without any rules built-in. Terrain size, steepness, vegetation density, creating small or big forests, having grass and plants near water and pines in higher altitudes (or vice versa) and anything the user wishes can be done. The decision to not include any built in rules was done in favour of having a completely customizable tool, able to generate any environments.

3.2 Use Case Scenarios Diagrams

In this section we provide the use case diagrams depicting the user's interaction with the tool.

3.2.1 User imports tool into Unity

The figure bellow depicts the initial process of using the tool once it is imported into Unity.

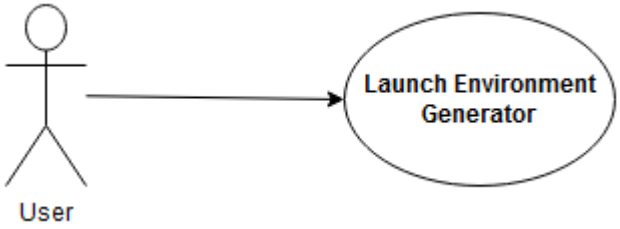

Figure 3.1: Initial Use Case.

Once the tool is imported into Unity, the user will be able to launch the custom window called "Environment Generator" thus having access to the available options for environment generation.

3.2.2 User has launched the Environment Generator

The figure bellow illustrates the interactions available after the user has launched the Environment Generator, and wants to start creating environments.

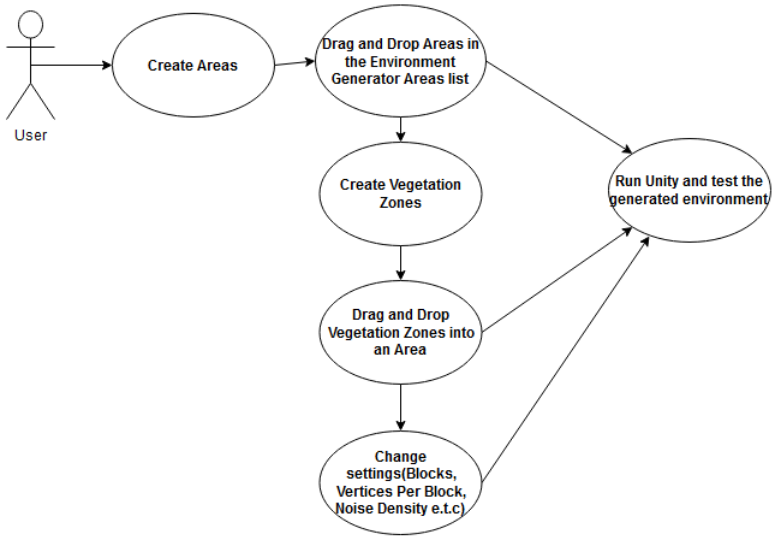

Figure 3.2: Environment Generator Options Use Case.

- **Create Areas:** The user can drag and drop the Area script onto empty game objects and adjust the settings of the script to create an Area.

- **Drag and Drop Areas in the Environment Generator Areas list:** In order for the data of any areas the user has created to be used by the Generator, they must be drag and dropped into the available list.

- **Create Vegetation Zones:** The user can drag and drop the Vegetation Zone script onto empty game objects and adjust the settings of the script to create an Vegetation Zone.

- **Drag and Drop Vegetation Zones into an Area:** In order for the data of any vegetation zones the user has created to be used by the Generator, they must be drag and dropped into the available list of the desired area.

- **Change settings(Blocks, Vertices Per Block, Noise Density e.t.c):** The user should adjust the settings according to their preferences. These settings regard the terrain's size and shaping.

- **Run Unity and test the generated environment:** The user can run Unity at any point to see the results of their chosen settings.

3.2.3 User is running Unity

The figure bellow illustrates the interactions available after the user has ran Unity and generated an environment.

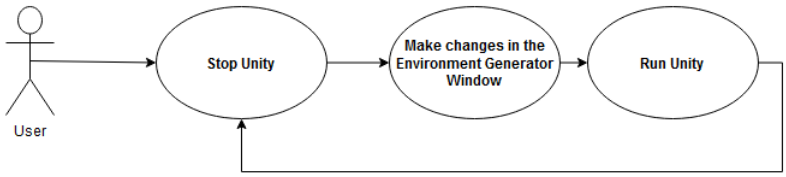

Figure 3.3: Unity running use case.

Once Unity is running, if the user wishes to make any changes, they must first stop Unity then make any changes in the Environment Generator and finally re run Unity to see the changes.

In the sections following this one, we dive deeper into explaining each case more carefully.

3.3 Creating an Environment Generator

Assuming the user has imported this tool into their Unity project, an Environment Generator should be created initially. To make this part simpler, a custom Editor Window containing all the necessary fields to generate an environment was created using the UnityEditor library. This library provides an API to help developers create custom windows and layouts that fit their project.

After importing the tool, a new menu tab (figure 3.4) should be available called "PCG" that gives access to the Environment Generator as shown in the figure below. After launching the generator, a custom window will pop up (figure 3.5), containing various inputs that will affect the generated terrain. Finally, a new game object called "BlockGenerator" should appear, containing a BlockGenerator and EnvironmentController script (figure 3.6).

All of the fields seen in the Environment Generator window are internally linked with variables in the BlockGenerator script that are used to generate the environment. More specifically, the values that get set in the environment window are also set automatically inside the program in the corresponding variables.

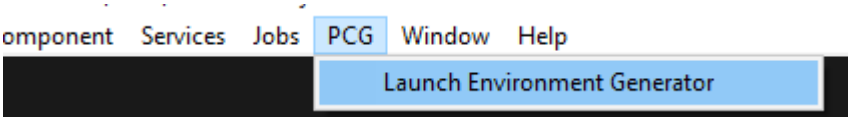

Figure 3.4: Menu tab of the custom Editor Window for the Environment Generator.

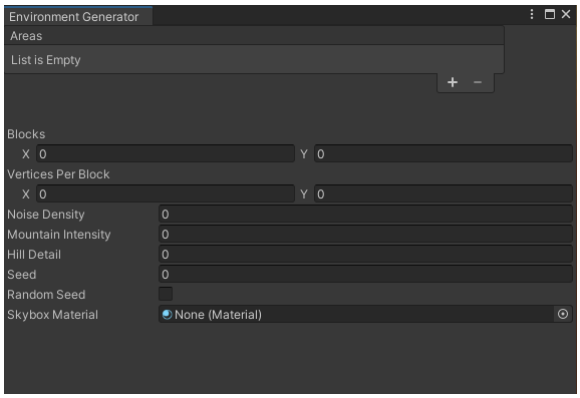

Figure 3.5: Custom Editor Window for the Environment Generator.

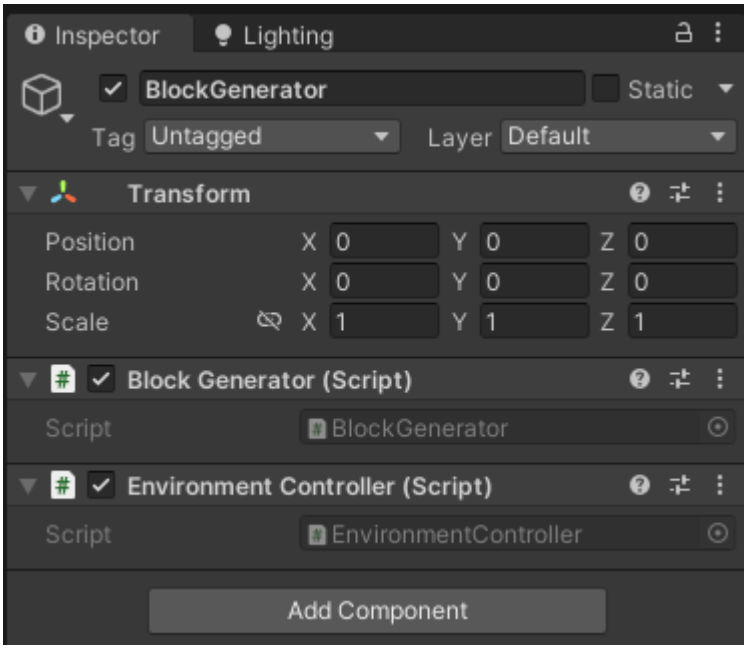

Figure 3.6: The BlockGenerator game object that gets created, containing the appropriate scripts.

3.3.1 Field Explanation

Let's start explaining each field. We will leave the Areas list for later. The Vector2 field Blocks is the terrain's size. More specifically, the algorithm begins by creating X times Y game objects and placing them 128 units apart from each other both horizontally and vertically. These game objects are called "Blocks". Each Block has a Mesh Renderer along with a Mesh Filter to render the mesh as well as a Mesh Collider to handle collisions detection. Finally, it has a TerrainGenerator script attached which is responsible for the shaping of the mesh with noise (figure 3.8).

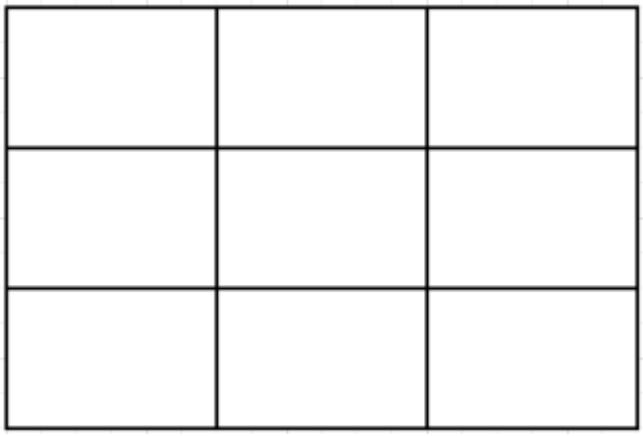

Figure 3.7: An example 3x3 grid after the first step of the algorithm.

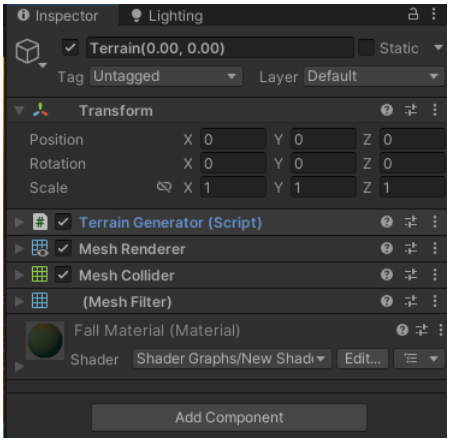

Figure 3.8: Block game object.

Moving on to the Vertices Per Block Vector2 field, it regards the X and Y vertices each

Block's mesh has. On these vertices, vegetation spawns, meaning more vertices give denser vegetation.

Below the Vertices Per Block field, we can see three fields that hold float values: Noise Density, Mountain Intensity and Hill Detail. These three float values are multipliers in the code, that modify the noise map produced. To be more specific, Noise Density affects the base noise that gets produced, making it denser as we increase it. For a first example, let's consider a 16x16 terrain, with a singular area covering all of the map with no vegetation zones (so we can focus on the terrain shaping) and all of the values mentioned above at 0:

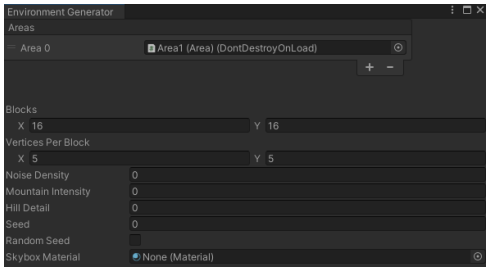

Figure 3.9: First example values.

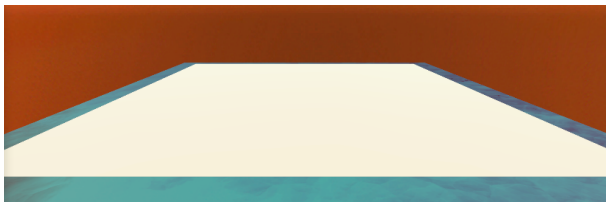

Figure 3.10: First example: terrain result.

As we can clearly see, the terrain that gets produced is a perfectly flat surface. That is due to the multipliers all being zero, thus every Noise function output gets set to 0. The Noise Density field is the most important multiplier because it enhances the basic noise plate that gets created internally. Let's see the result by setting the Noise Density field to 150:

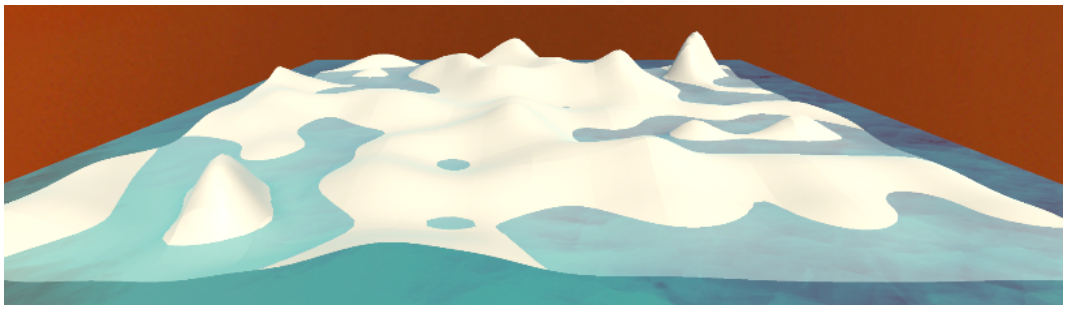

Figure 3.11: First example: terrain result, Noise Density = 150.

By this small adjustment, we can already see big differences with small hills and valleys being created. Now let's test some Mountain Intensity values below:

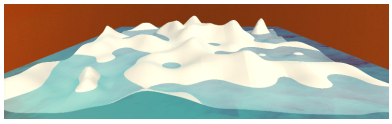

Figure 3.12: Noise Density = 150, Mountain Intensity = 150.

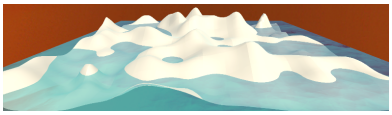

Figure 3.13: Noise Density = 150, Mountain Intensity = 250.

Figure 3.14: Noise Density = 150, Mountain Intensity = 400.

By comparing the initial terrain shown in the figure 3.8, by increasing the Mountain Intensity multiplier the mountains are getting taller and the valleys shallower. For example, the bottom left island slowly disappears and the upper right mountains are getting taller. However all of the terrain is way too rounded and smooth. That is where the third multiplier, Hill Detail comes into use.

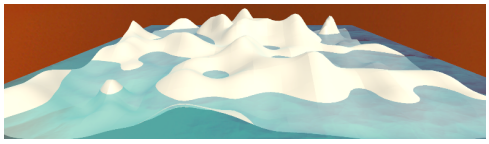

Figure 3.15: Noise Density = 150, Mountain Intensity = 250, Hill Detail = 0.

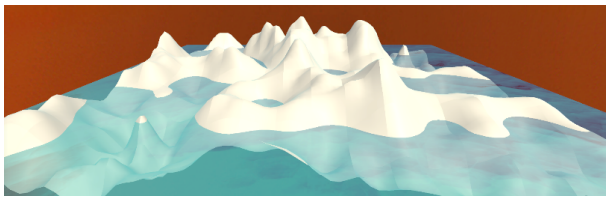

Figure 3.16: Noise Density = 150, Mountain Intensity = 250, Hill Detail = 300.

In this last comparison, we can clearly see that all of the terrain is less smooth and has some extra texturing, making it much more realistic. Of course, all of these examples are with a base white material and no vegetation, meant to show off what these three fields are capable of creating.

The Seed integer field and the Random Seed boolean field are related. Since much of the program and the noise functions use the random number generator, we must understand how it works. For starters, the numbers generated are not truly random. Instead they are produced in a preset sequence and the seed value is the point where a particular sequence begins. By default, the seed is selected by the system. However, there are instances where we want to choose the seed. In our tool for example, we can generate terrains with random seeds by having the Random Seed boolean checked. This would create different terrains on each run. As we randomly generate terrains though, we might find a particular one that we like. That means

that the pseudo-random values that we randomly got, suit our needs and those specific values have a specific seed. Thus by feeding the specific seed into the system, we can keep getting the same terrain over and over again. To use a seed we want, we uncheck the Random Seed boolean, and enter the seed value into the Seed integer field. For example, the figures 3.8-3.13 all had the seed value set at 0, and that is why the same terrain was generated, allowing us to see the differences when adjusting the three multipliers. Let's see an example, with the same multipliers but Random Seed checked.

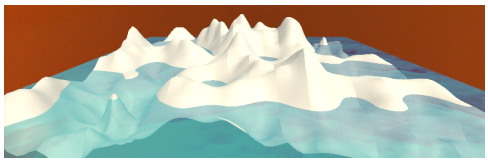

Figure 3.17: Noise Density = 150, Mountain Intensity = 250, Hill Detail = 300, Seed = 0.

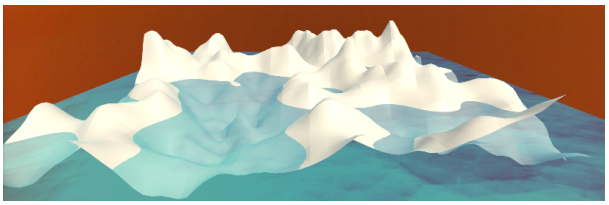

Figure 3.18: Noise Density = 150, Mountain Intensity = 250, Hill Detail = 300, Random Seed.

As we can see, the same multipliers give different results due to the random seed, although there are still some similarities.

Finally, the Skybox game object field is used in case the user wished to have a custom skybox instead of Unity's default one.

3.3.2 Areas

Areas is the system built in this thesis to allow multiple different sub-environments to exist within the created world. Many games with massive worlds consist of different types of environments for the player to traverse through, so the addition of a similar system is implemented. To create a custom area, the user has to create an empty game object and attach the "Area" script on to it (figure 3.19). Let's begin explaining the simpler fields first.

The Zone Starting/Ending Points X/Z set the limits of the area in the X and Z axis. Both X and Z limits of the terrain are calculated by multiplying 128 with the Environment Generator's window Block field, in the X and Y respectively, minus one. For example, if we generate a 16x16 terrain, the limits of X and Z are from 0 to 1920 (0 * 128 to 15 * 128). The material field is used to apply any material to the Blocks that are included withing the limits given by the user.

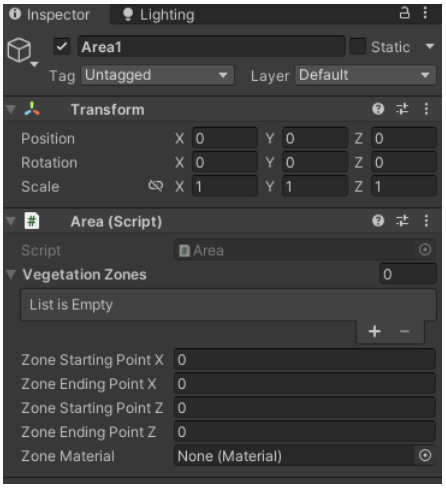

Figure 3.19: Empty game object with Area script attached.

Each Area can also have their own vegetation zones. Vegetation zones split the area vertically, spawning any vegetation given within the height limits that are set. In order to create a Vegetation Zone, the user has to create an empty game object and attach the VegetationZone script to it, similar to creating Areas (figure 3.20).

The Zone Starting/Ending point Y refers to the limits on the Y axis that the vegetation placed in the lists above will spawn. The system considers the water level as zero, so the spawn points of vegetation span between (waterLevel + ZoneStartingPointY) to (waterLevel + ZoneEndingPointY). Moreover, the last field named **AmountOfTreeFriends** allows the user to determine how many smaller plants will spawn around each tree.

As mentioned before, vegetation spawns on the vertices of each block. Without diving deeper in this chapter, the algorithm iterates through each block's vertex and will decide if it will spawn vegetation or not depending on the probability given by the user through the Vertex Spawn Probability slider.

The two lists regard vegetation itself. Weighted Zone Trees list contains the 3D models and prefabs of trees that the user wishes the specific zone to have and their corresponding weights. Weighted Zone Tree Friends list contains the possible smaller vegetation such as grass or flowers that will spawn around each tree and their corresponding weights.

Finally, to understand the Tree Spawn Prob slider, let's discuss a bit about how vegetation spawns. Initially, the algorithm passed through each vertex and based on the Vertex Spawn Prob, decides if it will spawn any vegetation or not. If it decides to spawn vegetation, it chooses

one 3D model to place exactly on the vertex and then surrounds said 3D model with the prefabs in the Weighted Zone Tree Friends list. The central 3D model that spawns on the vertex can be selected from both of the lists. Assuming the user has trees in the first list and smaller vegetation in the second list, this option is given so that any vegetation zone can be populated with more trees, more smaller vegetation or a mix, depending on the user's needs. By having the Tree Spawn Prob at 1, the program will only spawn 3D models from the Weighted Zone Trees on the vertices and surround them with the 3D models in the Weighted Zone Tree Friends list. If the Tree Spawn Prob is at 0, the program will spawn 3D models from the Weighted Zone Tree Friends list and surround them with 3D models from the same list.

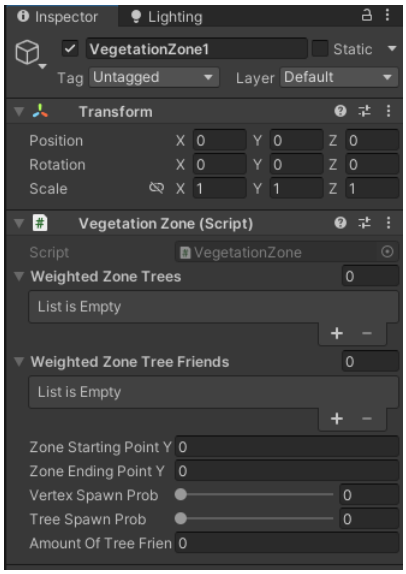

Figure 3.20: Empty game object with VegetationZone script attached.

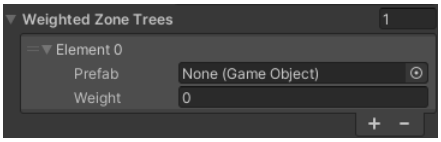

Figure 3.21: Weighted Zone Tree list element.

Let's see an example to better understand this. First we create one singular area covering the whole 16x16 terrain, with a custom material that we have created, one vegetation zone and drag the game object that it is attached to in the list of areas of the Environment Generator (figure 3.22). For the vegetation zone we will use one tree and one tree friend. Both of the

3D models were picked from the Unity Asset Store. We set the Vertex Spawn Prob at 0.5 so each vertex has 50 percent chance to spawn any vegetation, and the Tree Spawn Prob at 1 so the model that spawns on the vertex is always picked from the Weighted Zone Trees list (figure 3.23).

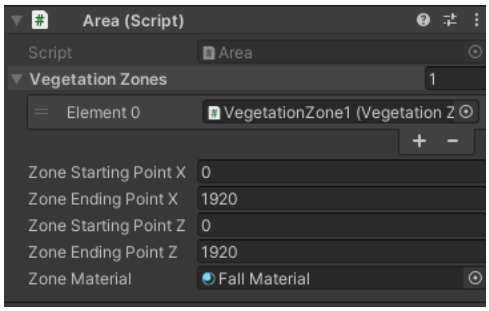

Figure 3.22: First area example.

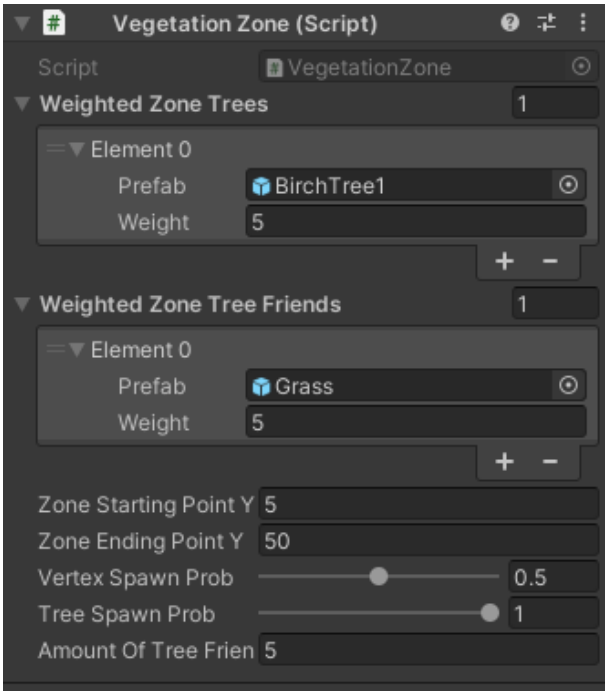

Figure 3.23: First vegetation zone example.

Figure 3.24: Tree prefab for the first example of vegetation zone.

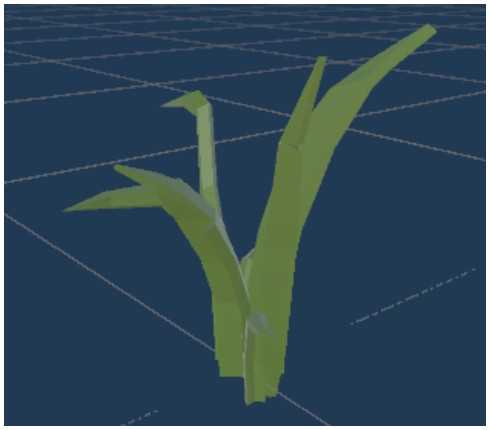

Figure 3.25: Grass example for the first example of vegetation zone.

As we can see the environment got filled with trees as the main vegetation with grass surrounding it, as we can see in the figures below.

Figure 3.26: Far away shot of vegetation spawned.

Figure 3.27: Birch Tree spawn on vertex with grass surrounding it.

Now if we set the Tree Spawn Prob at 0, we will only see patches of grass spawning. This will happen because on each vertex it will always choose the grass from the Weighted Zone Tree Friends list and spawn additional grass around it. This whole mechanic is used if the user wants for example to have only smaller vegetation near water, or taller trees only in higher altitudes, or any mixture they wish.

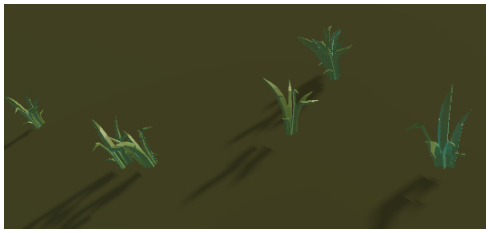

Figure 3.28: Grass patch spawn.

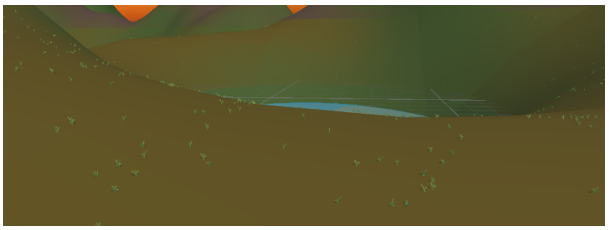

Figure 3.29: Far away shot of vegetation spawned.

Of course the environment generated in these images is not anything to brag about. Instead it is meant to show off the uses of the fields explained in this chapter. We will see more example of better looking terrain in another chapter.

Chapter 4

Implementation

4.1 Introduction

In this section a more detailed description of the thesis implementation will be provided. To be more specific, the first part will explain how the terrain is generated and shaped, how the scripts and the components are connected and then proceed with the explanation of the Areas logic and how the vegetation spawns.

4.2 Overall structure

In this section a description of the overall system's structure will be provided, how the components interact with each other and what is their purpose.There are three main components that interact with each other: BlockGenerator, TerrainGenerator and EnvironmentController as shown in the figure 4.1.

The basic flow of information between the components is:

- The BlockGenerator generates game objects called Blocks with TerrainGenerator, MeshRenderer, MeshFilter and MeshCollider components and places them 128 units apart in the X and Z axes.
- The TerrainGenerator on each Block generates its Mesh and shapes it with noise, applying the correct material based on its location and the area it belongs.
- The EnvironmentController spawns vegetation based on each vertex's location and the vegetation zone it belongs.

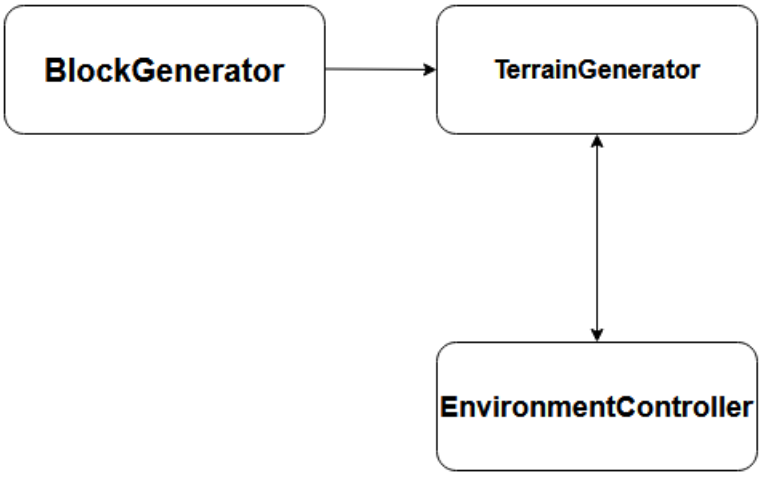

Figure 4.1: Structure of system.

The next subsections will dive deeper in each component and describe how they were implemented.

4.3 Block Generator

In this section, we will provide an explanation of how the BlockGenerator component works. This component is the simplest of the three and the starting point of the algorithm. Moreover, it contains all of the necessary variable fields that are required for the tool to work and it is directly connected with the Environment Generator Editor Window.

To begin with, we need to mention that this and the EnvironmentController component have Singleton Patter implemented. Singleton is a design pattern that turns a class into a data container, accessible by other classes in the project. The reason we implemented this design pattern is so that the three components can communicate with each other, without the need to create references of one another in each script. To see how the pattern was implemented, first we need to talk about the Awake method.

BlockGenerator script derives from MonoBehaviour. MonoBehaviour is a class containing various methods like Start, Update, Awake. Start and Awake functions are called once in the lifetime of the script. The Start function gets called on the first frame that a script gets enabled. The Awake function is called earlier than the Start function, when a script gets loaded.

In the figure 4.2 we can see the Awake function of the BlockGenerator script. It contains the

functionality of the seed mechanic as well as the Singleton design pattern. Regarding the latter, the script contains a variable called Instance of type BlockGenerator and it is an instance of the class itself. The instance is also static since we want it to be accessible globally. To implement the design pattern we check if we already have a reference to the instance variable and if the reference is not the one we are currently working with, meaning that there is another instance running. In that case, we destroy the current game object. This means that there can only be one BlockGenerator instance running. If none of these checks are true, then we set the instance to the current one.

In the figure 4.4 we can see the Start function of the BlockGenerator script. It starts by generating a random value that is used to give a greater element of randomness to the noise generation of the TerrainGenerator scripts. It also sets the skybox material, if one is placed in the Custom Editor window. Finally, we instantiate the water through the **InstantiateWater** function, call the **GenerateBlocks** coroutine and randomizes the mountain height of the generated environment.

Moving on to the seed mechanic. The script has a variable named **seed** of type **integer** that holds the possible number the user can enter to use as a seed. Additionally, another variable named **randomSeed** of type **boolean** is used to check whether the user wants to use a random seed or a specific one. If the boolean is checked, then the program will use the number saved in the seed variable to generate random numbers using the Random static class Unity provides. Inside the Awake function, we check if the randomSeed boolean is not checked, in which case we use the Random's class function InitState to initialize the random number generator's state with the seed given by the user.

```
private void Awake()
{
    //Check if randomSeed is not checked, meaning the user wants to use a specific seed.
    if(!randomSeed)
    {
        Random.InitState(seed); //Initialize random number generator's state with specific seed.
    }

    //Singleton Pattern
    if (Instance != null && Instance != this) //Check if instance already exists, else create one.
    {
        Destroy(gameObject);
    }
    else
    {
        Instance = this;

    }
}
```

Figure 4.2: BlockGenerator's Awake function.

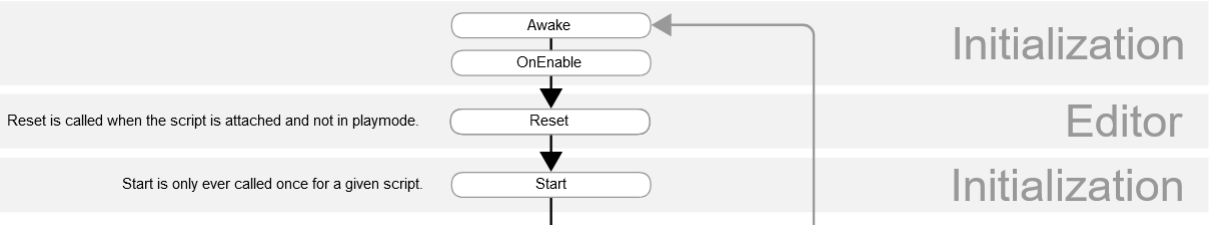

Figure 4.3: Order of execution for event functions.

```
private void Start()
{
    randomValue = Random.Range(-100, 500);

    if (skyboxMaterial != null)
        RenderSettings.skybox = skyboxMaterial;

    InstantiateWater();
    mountainHeight = Random.Range(5f, 10f);
    StartCoroutine(GenerateBlocks());
}
```

Figure 4.4: BlockGenerator's Start function.

Moving on, we will talk about the **GenerateBlocks** coroutine. In general, when a method is called, it runs until it is finished and then the program continues from the point the method was called. A coroutine is a method that allows a sequence of events to be completed over time and switching control between the main program and itself across several frames. It is declared with an IEnumerator return type and must contain a yield return statement, which declares the point that the coroutine pauses and resumes later. Regarding the **GenerateBlocks** coroutine, it begins by creating a child of the game object that the script is attached to whose job is to contain all the block game objects that will be generated. This child is a variable named **BlockCollection** of type **GameObject** and it is used to keep the Hierarchy tab of Unity clean and not fill it with hundreds of game objects. The main body of the coroutine is the generation and placement of block game objects. This is done with a nested for loop. Both of the loops start at 0 and their conditions depend on a variable named **blocks** of type **Vector2**. To be more specific, the outside loop runs from 0 to the X component of the Vector2 and the nested loop from 0 to the Y component of the Vector2, both increment by 1 on each iteration. The **blocks** variable is directly linked with the **Blocks** field on the Custom Editor Window, and this is the way that the size of the generated terrain gets determined. On each iteration of the loops, a new Block is created, set as a child of BlockCollection and placed in the correct

position. Finally, there is a variable named **batchCounter** of type **integer** that speeds up the block generation by spawning them in batches of 32 and not one by one. The batchCounter variable is increased by one on each iteration, and once it reaches 32, the yield statement is called with WaitForSeconds which suspend the coroutine for a couple of frames.

Finally, we have the **InstantiateWater** function of return type **void** that instantiates the water prefab with a random offset as it can be seen in figure 4.6.

```
//Terrain is a set of blocks, method to generate them and assign the TerrainGenerator script.
public IEnumerator GenerateBlocks()
{
    //Organize blocks in this game object's first child.
    if (transform.childCount == 0)
    {
        GameObject blockCollection = new GameObject("BlockCollection");
        blockCollection.transform.parent = this.transform;
    }

    for (int batchCounter = 0,  i = 0; i < blocks.x; i++)
    {
        for(int j = 0; j < blocks.y; j++)
        {
            //Generate block and place it
            GameObject block = new GameObject("Block" + new Vector2(i * 128, j * 128),typeof(TerrainGenerator),typeof(MeshRenderer),typeof(MeshCollider),typeof(MeshFilter));

            block.transform.parent = transform.GetChild(0).transform; //Set BlockCollection as the parent of each Block.
            block.transform.position = new Vector3(i * 128, 0f, j * 128);//Place the Block correctly.

            batchCounter++;

            if (batchCounter == 32)//Generate block in batches/
            {
                batchCounter = 0;
                yield return new WaitForSeconds(Time.deltaTime);
            }
        }
    }
}
```

Figure 4.5: BlockGenerator's GenerateBlocks coroutine.

```
//Method to instantiate water
 private void InstantiateWater()
{
    float waterLevelOffset = Random.Range(-15f, -60f);
    waterLevel = Mathf.PerlinNoise(randomValue, randomValue) * waterLevelOffset;
    GameObject wat = Instantiate(water, new Vector3((128 * blocks.x) / 2, waterLevel, (128 * blocks.y) / 2), Quaternion.identity);
    wat.transform.localScale = new Vector3(3 * blocks.x, 0f, 3 * blocks.x);
}
```

Figure 4.6: BlockGenerator's InstantiateWater function.

4.4 Terrain Generator

In this section, we will provide an explanation of how the TerrainGenerator component works. This component is attached on every Block that the BlockGenerator component creates. Its purpose is primarily to create and shape the mesh of each block using Perlin Noise. This class contains the following variables:

- **mesh** of type Mesh.
- **meshFilter** of type MeshFilter.
- **meshRenderer** of type MeshRenderer.
- **meshCollider** of type MeshCollider.
- **UVs** of type Vector2[].
- **vertices** of type Vector3[].
- **triangles** of type int[].
- **rend** of type Renderer.

The last variable helps render the correct material on each block while all the others are responsible for creating the mesh. Meshes have different type of data such as vertices, triangles and UVs, which the variables mentioned above are used for. UVs are used to correctly apply materials to meshes. The field **VerticesPerBlock** of the Custom Editor Window is used to set the vertices of each Block mesh and is directly linked with the variable **verticesPerBlock** of type **Vector2** in the BlockGenerator class. The size the Vector3 array of vertices mentioned above is **(BlockGenerator.Instance.verticesPerBlock.x + 1)*(BlockGenerator.Instance.verticesPerBlock.y + 1)** (the +1 is for array indexing).

Now let's start explaining the major function of this class called **GenerateTerrain** of return type **void** (figures 4.7, 4.8). This function's use is to place all the vertices correctly depending on how many the user enters and shape the mesh using Perlin Noise. It consists of a nested for loop, going from 0 up to the verticesPerBlock input of the user, placing each vertex on the X and Y axes of each Block mesh. For the shaping to occur, essentially the Y position of the vertex is calculated through Perlin Noise. This Perlin Noise calculation is done by two helper functions: **Noise** and **BiomeNoise** both of return type **float**. These

two functions will be explained later. Following the nested loop for the vertices is a simple loop to assign the texture coordinates in the UVs array. Finally, another nested for loop is used to fill the triangles array. Triangles need three vertices to be described and drawn. In computer graphics there is a process known as Back Face Culling, which is used to eliminate the back sides of triangles. So, in order for triangles to be drawn they must be facing the right direction and to determine that, the order in which the points are fed in the program is taken into account. Specifically in Unity, triangles are drawn clockwise, so we must feed the vertices in a clockwise direction for Unity to draw them correctly. Since each quad that gets created by two triangles with three points each, the size of the triangles array is **(BlockGenerator.Instance.verticesPerBlock.x*BlockGenerator.Instance.verticesPerBlock.y*6)**. Finally, the now filled with data arrays get set in the mesh's data. Continuing on the explanation of the TerrainGenerator component, we will explain the two helper functions **Noise** and **BiomeNoise**. To create Perlin Noise, we used Unity's **PerlinNoise** function provided by the **Mathf** library which contains a collection of math functions. Mathf's PerlinNoise function takes two floats x and y as inputs and returns a float value between 0.0 and 1.0. When called, essentially a pseudo-random pattern of float values is generated, and by inputting specific X and Y coordinates we get a specific sample of that pattern (figure 4.9).

```
//Method to generate and shape mesh.
void GenerateTerrain()
{
    mesh = new Mesh();
    vertices = new Vector3[(int)((BlockGenerator.Instance.verticesPerBlock.x + 1)*(BlockGenerator.Instance.verticesPerBlock.y + 1))];
    UVs = new Vector2[vertices.Length];

    //Count vertices.
    int vertexIndex = 0;

    //Loop through both X and Y vertices of each mesh.
    for (int i = 0; i <= BlockGenerator.Instance.verticesPerBlock.x; i++)
    {
        for(int j = 0; j <= BlockGenerator.Instance.verticesPerBlock.y; j++)
        {
            float noiseValue = 0;

            noiseValue = Noise(i, j, BiomeNoise(i, j));//Generate noise value

            //Vertex location.
            vertices[vertexIndex] = new Vector3(i * (128 / BlockGenerator.Instance.verticesPerBlock.x), noiseValue, j * (128 / BlockGenerator.Instance.verticesPerBlock.y));
            vertexIndex++;

        }
    }

    //Calculate UVs
    for (int i = 0; i < UVs.Length; i++)
    {
        UVs[i] = new Vector2(vertices[i].x,vertices[i].z);
    }
```

Figure 4.7: First part of TerrainGenerator's GenerateTerrain function.

```
    //Need to multiply by 6 because each square consists of 2 triangles with 3 vertices.
    triangles = new int[(int)(BlockGenerator.Instance.verticesPerBlock.x*BlockGenerator.Instance.verticesPerBlock.y*6)];
    vertexIndex = 0;
    int triangleIndex = 0;

    //Calculate triangles
    for(int i = 0; i < BlockGenerator.Instance.verticesPerBlock.y; i++)
    {
        for(int j = 0; j < BlockGenerator.Instance.verticesPerBlock.x; j++)
        {
            triangles[triangleIndex] = (int)(vertexIndex + BlockGenerator.Instance.verticesPerBlock.x + 2);
            triangles[triangleIndex + 1] = (int)(vertexIndex + BlockGenerator.Instance.verticesPerBlock.x + 1);
            triangles[triangleIndex + 2] = vertexIndex + 1;
            triangles[triangleIndex + 3] = vertexIndex + 1;
            triangles[triangleIndex + 4] = (int)(vertexIndex + BlockGenerator.Instance.verticesPerBlock.x + 1);
            triangles[triangleIndex + 5] = vertexIndex;
            vertexIndex++;
            triangleIndex += 6;

        }
        vertexIndex++;
    }

    mesh.Clear();
    mesh.vertices = vertices;
    mesh.uv = UVs;
    mesh.triangles = triangles;
    mesh.RecalculateBounds();
    meshCollider.sharedMesh = mesh;
    mesh.RecalculateNormals(); //Used for better lighting.
    meshFilter.mesh = mesh;
}
```

Figure 4.8: Second part of TerrainGenerator's GenerateTerrain function.

Figure 4.9: Example pattern sample of PerlinNoise function.

The function **BiomeNoise** (figure 4.10) creates the base plate of noise for our terrain. Inside this function we use Mathf's PerlinNoise three times: one time to get a base plate, then we add some more Perlin Noise onto that to create mountains and finally subtract some to create valleys. We use each vertex's position as an input multiplied by a small value to get finer noise samples. These multipliers were determined with lots of trial and error.

```
//Calculate initial noise values
public float BiomeNoise(float x, float z)
{
    //Base noise plate, spread out.
    float noiseValue = Mathf.PerlinNoise(((x * (128 / BlockGenerator.Instance.verticesPerBlock.x)) + transform.position.x + BlockGenerator.Instance.randomValue) * 0.003f,
                                         ((z * (128 / BlockGenerator.Instance.verticesPerBlock.y)) + transform.position.z + BlockGenerator.Instance.randomValue) * 0.003f);
    //Add finer noise for mountains , hills e.t.c.
    noiseValue += Mathf.PerlinNoise(((x  * (128 / BlockGenerator.Instance.verticesPerBlock.x)) + transform.position.x + BlockGenerator.Instance.randomValue) * 0.001f,
                                    ((z * (128 / BlockGenerator.Instance.verticesPerBlock.y)) + transform.position.z + BlockGenerator.Instance.randomValue) * 0.001f);
    //Noise for valleys, oceans.
    noiseValue -= Mathf.PerlinNoise(((x * (128 / BlockGenerator.Instance.verticesPerBlock.x)) + transform.position.x + BlockGenerator.Instance.randomValue) * 0.00005f,
                                    ((z * (128 / BlockGenerator.Instance.verticesPerBlock.y)) + transform.position.z + BlockGenerator.Instance.randomValue) * 0.00005f) * 2f;
    //-1 oceans -> 1 mountains.
    noiseValue = Mathf.Clamp(noiseValue, -1, 1);

    return noiseValue;
}
```

Figure 4.10: TerrainGenerator's BiomeNoise function.

Moving on to the **Noise** (figure 4.11) function, it modifies and creates the final noise values that will be used. It uses the input of the BiomeNoise function mentioned above and the x and z position of the vertex. Initially, we use the given noise and multiply it by the **NoiseDensity** field of the Custom Editor Window, which is directly linked with the variable named **noiseDensity** of type **float** inside the BlockGenerator script. By multiplying the noise, we create steeper valleys and mountains since we get a bigger float output. That first step gives us the base plate for the terrain. The second step, using some simple math and after experimenting, focuses more on making the higher and lowest points of the noise steeper as well as adding more detailing on the hills. This is done by adding noise with different coefficients in the amplitude and frequency parts. Moreover, in the hills and mountains enhancement part of the code we make use of the **MountainIntensity** and **HillDetail** fields of the Custom Editor Window which are directly linked with the variables **mountainIntensity** and **hillDetail** both of type **float** inside the BlockGenerator script.

```
//Calculate final noise values
public float Noise(float x, float z, float noise)
{
    //Base plate
    float noiseValue = noise * BlockGenerator.Instance.noiseDensity;
    //Mountains
    float multiplier = 1 + Mathf.Pow(noise,3) * BlockGenerator.Instance.mountainHeight;
    noiseValue *= multiplier;

    noiseValue += (Mathf.PerlinNoise((((x + BlockGenerator.Instance.randomValue) * (128 / BlockGenerator.Instance.verticesPerBlock.x)) + transform.position.x) * 0.005f,
    ((z * (128 / BlockGenerator.Instance.verticesPerBlock.y)) + transform.position.z + BlockGenerator.Instance.randomValue) * 0.005f) * BlockGenerator.Instance.mountainIntensity) * noise;

    //Hills
    noiseValue += (Mathf.PerlinNoise((((x + BlockGenerator.Instance.randomValue) * (128 / BlockGenerator.Instance.verticesPerBlock.x)) + transform.position.x) * 0.009f,
    ((z * (128 / BlockGenerator.Instance.verticesPerBlock.y)) + transform.position.z + BlockGenerator.Instance.randomValue) * 0.009f) * BlockGenerator.Instance.hillDetail) * noise;

    return noiseValue;
}
```

Figure 4.11: TerrainGenerator's Noise function.

There are also two helper functions: **CheckIfEdgeVertex** of return type **bool** and **SetStartingColors** of return type **void**. The first one is used to check if a vertex is on the edge of the terrain that help us not spawn vegetation there that gets outside of the terrain and the second sets the material of each block by figuring out in which area it belongs and using the area's specific material.

```
private bool CheckIfEdgeVertex(Vector3 vertex)
{
    if (vertex.x == 0 || vertex.x == 128 || vertex.z == 0 || vertex.z == 128)
        return true;
    else
        return false;
}

//Set starting terrain colors based on which area the block belongs.
private void SetStartingColors()
{
    //Loop through areas.
    for(int i = 0; i < BlockGenerator.Instance.areas.Count; i++)
    {
        Area temp = BlockGenerator.Instance.areas[i];

        //Check if block belongs in area.
        if ((this.transform.position.x >= temp.zoneStartingPointX) && (this.transform.position.x <= temp.zoneEndingPointX) &&
            (this.transform.position.z >= temp.zoneStartingPointZ) && (this.transform.position.z <= temp.zoneEndingPointZ))
        {
            rend.material = temp.zoneMaterial;
        }
    }

}
```

Figure 4.12: TerrainGenerator's CheckIfEdgeVertex and SetStartingColors functions.

Finally, we need to explain the code in the Start function of the script. Initially we call a function named **Setup** of return type **void** that just gets the three mesh components of each Block. Then we call the GenerateTerrain function that shapes each block and call SetStartingColors to set the materials of each Block. Finally, we loop through each vertex of the Block to

decide on vegetation using the **GenerateVegetation** method that is inside the Environment-Controller and will be explained later. One final note for this section is the fact that we did not implement Singleton design pattern on the TerrainGenerator script since there is one attached on each Block so we need to have many of them running at the same time.

```
// Start is called before the first frame update
void Start()
{
    Setup(); //Get Mesh components of Block.
    GenerateTerrain(); //Shape Block.
    rend = GetComponent<Renderer>();
    SetStartingColors(); //Set Block material.

    //Loop through each vertex and decide on vegetation spawn.
    for (int i = 0; i < vertices.Length; i++)
    {
        if (vertices[i].y > BlockGenerator.Instance.waterLevel)
        {
            if (!CheckIfEdgeVertex(vertices[i]))
            {
                EnvironmentController.Instance.GenerateVegetation(vertices[i], transform.position.x, transform.position.z);
            }
        }

    }
}

void Setup()
{
    meshFilter = GetComponent<MeshFilter>();
    meshRenderer = GetComponent<MeshRenderer>();
    meshCollider = GetComponent<MeshCollider>();
}
```

Figure 4.13: TerrainGenerator's Start and Setup functions.

4.5 EnvironmentController

Moving on to the final main component of our program, this section will provide an explanation of the EnvironmentController script. This component has Singleton design pattern implemented (figure 4.15) and its main purpose is to handle the spawn of vegetation on each vertex. It directly communicates with the Areas created which themselves communicate with the Vegetation Zones. The script includes only one variable named **correctPos** of type **Vector3** that is used for the placement of tree friends spawn and will be explained later. The script begins with the implementation of the Singleton.

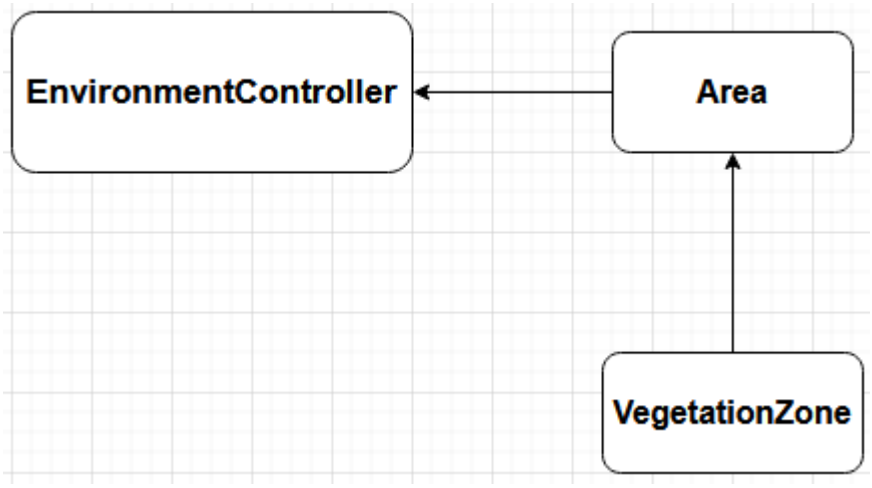

Figure 4.14: EnvironmentController's communication chart.

```
public static EnvironmentController Instance { get; set; }

private Vector3 correctPos;


private void Awake()
{
    if (Instance != null && Instance != this)
        {
            Destroy(gameObject);
        }
    else
        {
            Instance = this;
        }
}
```

Figure 4.15: EnvironmentController's Singleton implementation.

Following the Awake function, we find the **GenerateTreeFriend** coroutine(figures 4.16, 4.17. This coroutine is used to generate a tree friend somewhere around a specific vertex and it chooses the 3D model to spawn from the Weighted Zone Tree Friends list that each Vegetation Zone has. For a small reminder, the terrain is split into areas and each area has their own list of vegetation zones that split the area vertically. Each vegetation zone has a list of trees as well as a list of smaller vegetation like grass and plants that are called tree friends. Initially we get the area that the vertex belongs to and iterate through its vegetation zone list to determine in which one the vertex is placed. Once that is determined, we use a helper function named **GetRandomValue** of return type **GameObject** to get a random 3D model from the list. Each entry in both the Weighted Zone Trees and Weighted Zone Tree friends lists have a weight, meaning that the probability of each model spawning is higher when its weight is bigger. The GetRandomValue function uses simple math to return a GameObject as an output depending on the weight of each game object we have added on our list. After deciding which tree friend will spawn, we choose a random position around the main vertex's game object placement. However, while coding the spawn of tree friends, it was difficult to place it in the correct y position of the mesh. To achieve that, we used the help of **raycasts**. Raycast in Unity is a Physics function that projects a Ray into the scene, returning a boolean value if a target was successfully hit. When this happens, information about the hit, such as the distance, position or a reference to the object's Transform, can be stored in a Raycast Hit variable for further use. Essentially what is happening is that we spawn the 3D models higher up in the y axis and "shoot" a ray downwards. When that ray hits the ground, we get the correct y location that the game object needs to spawn in. We also get the correct rotation in the same way, depending on the slope of the terrain. In the raycast script (figure 4.18), the method **GetCorrectPosition** returns the Vector3 position that it hits and the method **GetCorrectRotation** returns the Quaternion rotation. The reason that GenerateTreeFriend is a coroutine and both the correct position and rotation are updated in the update method is because it takes a couple of frames for the raycast to get the correct data that we need, and not just one frame.

```
//Coroutine to generate a tree friend in a location around a tree.
IEnumerator GenerateTreeFriend(Vector3 vertex, GameObject tree, float posx, float posz)
{

    Area currentArea = GetVertexArea(vertex, posx, posz); //Get correct area.
    GameObject treeFriend = null;

    for (int i = 0; i < currentArea.vegetationZones.Count; i++) //Loop through vegetation zones.
    {
        if ((vertex.y > currentArea.vegetationZones[i].zoneStartingPointY) && (vertex.y < currentArea.vegetationZones[i].zoneEndingPointY)) //Get correct vegetation zone.
            treeFriend = GetRandomValue(currentArea.vegetationZones[i].WeightedZoneTreeFriends);
    }

    //If we succesfully chosen a specific tree friend, spawn it randomly somewhere around the main vertex and place it correctly on the mesh.
    if(treeFriend != null)
    {
        //Randomize the radius spawn.
        float radius = Random.Range(0.5f,15f);
        float scale = Random.Range(1f, 3f);

        Vector3 randomPos = Random.insideUnitSphere * radius;
        randomPos += tree.transform.position;

        //Spawn the 3D model higher and use raycasts to determine correct y placement on the mesh.
        if (vertex.y > 0)
            randomPos.y = 2f * vertex.y;
        else
            randomPos.y = -2f * vertex.y;

        //Add random rotation.
        float randomRotation = Random.Range(-360f, 360f);
```

Figure 4.16: EnvironmentController's GenerateTreeFriend function.

```
//Instantiate game object.
GameObject treeFriendSpawn = Instantiate(treeFriend, new Vector3(randomPos.x, vertex.y, randomPos.z), Quaternion.Euler(0, randomRotation, 0));

//Enable raycast script.
treeFriendSpawn.GetComponent<RaycastScript>().enabled = true;


treeFriendSpawn.transform.position = randomPos;
treeFriendSpawn.transform.localScale = new Vector3(scale, scale, scale);

//Wait for the ray to get the correct position
yield return new WaitForSeconds(2);

//Set to correct position.
treeFriendSpawn.transform.position = treeFriendSpawn.GetComponent<RaycastScript>().correctPos;

if (treeFriendSpawn.transform.position == Vector3.zero)
    Destroy(treeFriendSpawn);

treeFriendSpawn.transform.rotation = treeFriendSpawn.GetComponent<RaycastScript>().correctRot;

//Disable raycast script.
treeFriendSpawn.GetComponent<RaycastScript>().enabled = false;
//Set tree friend as child of the main 3D model spawned on the vertex.
treeFriendSpawn.transform.parent = tree.transform;
```

Figure 4.17: EnvironmentController's GenerateTreeFriend implementation continued.

```
// Update is called once per frame
void Update()
{
    correctPos = GetCorrectPosition();
    correctRot = GetCorrectRotation();
}

public Vector3 GetCorrectPosition()
{
    Ray ray = new Ray(transform.position, Vector3.down );
    Debug.DrawRay(ray.origin, ray.direction);
    Physics.Raycast(ray, out hitData);
    Vector3 hitPosition = hitData.point;
    return hitPosition;
}

public Quaternion GetCorrectRotation()
{
    Ray ray = new Ray(transform.position, Vector3.down);
    Debug.DrawRay(ray.origin, ray.direction);
    Physics.Raycast(ray, out hitData);
    Quaternion corrRot = Quaternion.FromToRotation(transform.up, hitData.normal) * transform.rotation;
    return corrRot;
}
```

Figure 4.18: Raycast script.

Moving on with the functions of the EnvironmentController component, the **GenerateVegetation** function of return type **void** uses another function called **GetVertexArea** of return type area to find in which area the specific vertex belongs to. Then, it figures out its vegetation area and calls the **GenerateTree** function. This function makes use of the two probabilities in the vegetation zone script: **VertexSpawnProb** and **TreeSpawnProb**. The first decides if anything will spawn on the vertex and the second determines from which of the two lists (Weighted Zone Trees and Weighted Zone Tree Friends) the main game object will get chosen. The function first decides if anything will spawn and then selects the main plant. Following that, it creates a child in the hierarchy to store all the main plants in order to keep it tidy and finally calls the GenerateTreeFriend function to spawn the number of tree friends the user wishes. In the function GetVertexArea, we make use of another helper function **VertexBelongsIn** of return type **bool** that checks if a vertex belongs in a specific area.

```
private void GenerateTree(Vector3 vertex, float posx, float posz,VegetationZone currentZone)
{
    //The probability to decide if something will spawn on the vertex.
    float spawnProbability = Random.Range(0f, 1f);

    GameObject tree;
    int amountOfTreeFriends = currentZone.amountOfTreeFriends;


    if (spawnProbability < currentZone.vertexSpawnProb)
    {
        GameObject whatToSpawn ;

        //The probability to decide if small plants or tree will spawn on the vertex as the central game object.
        float whatToSpawnProbability = Random.Range(0f, 1f);
        whatToSpawn = (whatToSpawnProbability > currentZone.treeSpawnProb) ? GetRandomValue(currentZone.WeightedZoneTreeFriends) : GetRandomValue(currentZone.WeightedZoneTrees)  ;

        whatToSpawn.GetComponent<RaycastScript>().enabled = false;

        //Spawn game object.
        tree = Instantiate(whatToSpawn,
        new Vector3(vertex.x + posx, vertex.y, vertex.z + posz),
        Quaternion.identity);
```

Figure 4.19: EnvironmentController's GenerateTree function.

```
//Create child to store all central game objects in order to keep hierarchy tidy
if(BlockGenerator.Instance.transform.childCount == 1)
{
    GameObject treeCollection = new GameObject("TreeCollection");
    treeCollection.transform.parent = this.transform;
}

tree.transform.parent = BlockGenerator.Instance.transform.GetChild(1).transform;

//Randomize rotation
int randomRotation = Random.Range(-360, 360);
tree.transform.rotation = Quaternion.Euler(0,randomRotation,0);

//Spawn tree friends.
for (int k = 0; k < amountOfTreeFriends; k++)
{
    StartCoroutine(GenerateTreeFriend(vertex, tree, posx, posz));
}
```

Figure 4.20: EnvironmentController's GenerateTree function continued.

```
//Method to get the area that a specific vertex belongs to.
private Area GetVertexArea(Vector3 vertex,float posx, float posz)
{
    int numberOfAreas = BlockGenerator.Instance.areas.Count;

    for (int i = 0; i < numberOfAreas; i++)
    {

        if (VertexBelongsInArea(vertex, BlockGenerator.Instance.areas[i],posx,posz))
        {
            return BlockGenerator.Instance.areas[i];
        }

    }

    return null;
}
```

Figure 4.21: EnvironmentController's GetVertexArea function.

```
//Helper method to check if a vertex belongs in a specific area.
private bool VertexBelongsInArea(Vector3 vertex, Area area, float posx, float posz)
{
    if ((vertex.x + posx > area.zoneStartingPointX && vertex.x + posx < area.zoneEndingPointX) && (vertex.z + posz > area.zoneStartingPointZ && vertex.z + posz < area.zoneEndingPointZ))
    {
        return true;
    }

    return false;

}
```

Figure 4.22: EnvironmentController's VertexBelongsInArea function.

```
//Method to generate vegetation based on the vertex's area and vegetation zone.
public void GenerateVegetation(Vector3 vertex, float posx, float posz)
{

    Area currentArea = GetVertexArea(vertex, posx, posz);

    if (currentArea != null)
    {
        for(int i = 0; i < currentArea.vegetationZones.Count; i++)
        {
            if((vertex.y > currentArea.vegetationZones[i].zoneStartingPointY) && (vertex.y < currentArea.vegetationZones[i].zoneEndingPointY))
                    GenerateTree(vertex, posx, posz, currentArea.vegetationZones[i]);
        }

    }
}
```

Figure 4.23: EnvironmentController's GenerateVegetation function.

4.6 Miscellaneous Implementations

In this section, we will provide information on various scripts such as the vegetation zone and area scripts and the implementation and testing of the Diamond Square algorithm that was not used in the final project. Let's start by listing the variables of the Area and Vegetation Zone scripts.

For the Area script:

- **vegetationZones** of type List.
- **zoneStartingPointX** of type float.
- **zoneEndingPointX** of type float.
- **zoneStartingPointZ** of type float.
- **zoneEndingPointZ** of type float.
- **zoneMaterial** of type Material.

```
public class Area : MonoBehaviour
{
    public List<VegetationZone> vegetationZones;
    public float zoneStartingPointX;
    public float zoneEndingPointX;
    public float zoneStartingPointZ;
    public float zoneEndingPointZ;
    public Material zoneMaterial;
```

Figure 4.24: Area script.

For the Vegetation Zone script:

- **WeightedZoneTrees** of type List.
- **WeightedZoneTreeFriends** of type List.
- **zoneStartingPointY** of type float.
- **zoneEndingPointY** of type float.
- **vertexSpawnProb** of type float (slider).
- **treeSpawnProb** of type float (slider).
- **amountOfTreeFriends** of type int.

```
public class VegetationZone : MonoBehaviour
{
    public List<WeightedValue> WeightedZoneTrees;
    public List<WeightedValue> WeightedZoneTreeFriends;
    public float zoneStartingPointY;
    public float zoneEndingPointY;
    [Range(0f, 1f)]
    public float vertexSpawnProb;
    [Range(0f, 1f)]
    public float treeSpawnProb;
    public int amountOfTreeFriends;
```

Figure 4.25: Vegetation Zone script.

For the Weighted Value script that is used to attach a game object with a specific weight:

- **prefab** of type GameObject.
- **weight** of type int.

```
[Serializable]
public class WeightedValue
{
    public  GameObject prefab;
    public int weight;
}
```

Figure 4.26: Weighted Value script.

Moving on we will briefly mention and show the implementation of the Diamond Square algorithm without going into too much detail since the idea was scrapped. We will also show the resulting terrain so that it is understood why it was not a correct fit for this thesis. First, let's remind the steps of the algorithm:

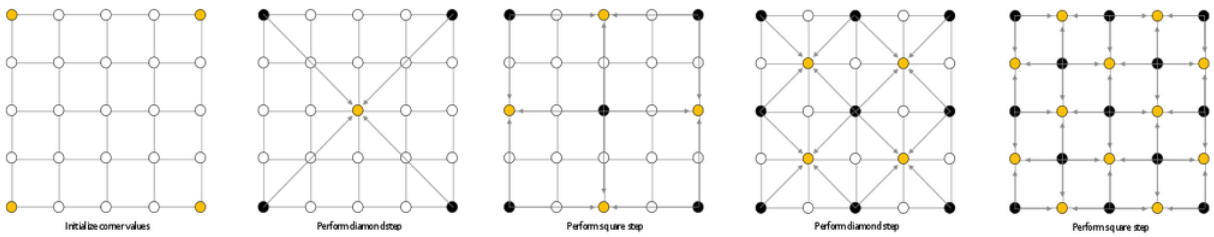

Figure 4.27: Diamond Square steps.

The script contains the following variables:

- **squares** of type int.
- **terrainSize** of type float.
- **maxHeight** of type float.
- **vertices** of type Vector3[].
- **UVs** of type Vector2[].
- **vertexCount** of type int.
- **triangles** of type int[].

The variable **squares** references the number of divisions or the faces we have (4 x 4 in the figure 4.27). For the first function of the script named **GenerateVerticesAndTriangles** of return type **void** we use this variable in two nested loops for the x and y axes. For the vertices we use 1D array to store them row by row. This function essentially generates the mesh in a similar way that TerrainGenerator component implements it (figure 4.28).

Moving on, the function named **DiamondAndSquareStep** of return type **void** performs one diamond and one square step. The diamond square, uses the values of the four corners to calculate the middle vertex value followed by the square step to calculate the four values located horizontally and vertically of the vertex (figure 4.29).

Finally, we have the functions **DiamondSquareAlgorithm** (figures 4.30, 4.31) and **GenerateTerrain** (4.32). The first one performs the Diamond Square algorithm by initiating the height values of the four corner points using Unity's Random generator and multiplying it by the **maxHeight** variable. Following this first step, we have a triple nested for loop: one for the iterations that the algorithm needs to perform the Diamond and Square step. If we take the example in figure 4.27 we can see that it takes two diamond and two square steps to calculate every vertex value. Essentially, it takes ($\log_2$ of the number of faces) iterations to fill every vertex. The other two loops iterate through rows and columns. Finally the **GenerateTerrain** functions puts everything together, generating the mesh and shapes it using the algorithm.

```
//Method to generate mesh.
void GenerateVerticesAndTriangles()
{
    float halfTerrain = 0.5f * terrainSize;
    float squareSize = terrainSize / squares;
    int triangleIndex = 0;


    for (int i = 0; i <= squares; i++)
    {

        for (int j = 0; j <= squares; j++)
        {
            vertices[i * (squares + 1) + j] = new Vector3(-halfTerrain + j * squareSize, 0.0f, halfTerrain - i * squareSize); //1D array holds all vertices, build row by row.
            UVs[i * (squares + 1) + j] = new Vector2((float)i/squares, (float)j/squares);

            if (i < squares && j < squares)
            {
                int topLeftVertex = i * (squares + 1) + j;
                int bottomLeftVertex = (i + 1) * (squares + 1) + j;


                //Create half upper right triangle of each square
                triangles[triangleIndex] = topLeftVertex;
                triangles[triangleIndex + 1] = topLeftVertex + 1;
                triangles[triangleIndex + 2] = bottomLeftVertex + 1;
                //Create half bottom left triangle of each square
                triangles[triangleIndex + 3] = topLeftVertex;
                triangles[triangleIndex + 4] = bottomLeftVertex + 1;
                triangles[triangleIndex + 5] = bottomLeftVertex;

                triangleIndex += 6;

            }
        }
    }
}
```

Figure 4.28: Diamond Square mesh generation.

```
//Method to perform diamond, followed by square step.
void DiamondAndSquareStep(int row, int column, int sizeOfSquare, float randomRange)
{
    int halfSize = (int)(sizeOfSquare*0.5f);

    //Diamond Step
    int topLeftVertex = row * (squares + 1) + column;
    int bottomLeftVertex = (row + sizeOfSquare)*(squares + 1) + column;
    int middleVertex = (int)(row + halfSize) * (squares + 1) + (int)(column + halfSize);

    vertices[middleVertex].y = (vertices[topLeftVertex].y + vertices[topLeftVertex + sizeOfSquare].y
                            + vertices[bottomLeftVertex].y + vertices[bottomLeftVertex + sizeOfSquare].y)/4 + Random.Range(-randomRange, randomRange); //Calculate mid point.

    //Square step
    vertices[topLeftVertex + halfSize].y = (vertices[topLeftVertex].y + vertices[topLeftVertex + sizeOfSquare].y + vertices[middleVertex].y)/3 + Random.Range(-randomRange, randomRange);
    vertices[middleVertex - halfSize].y = (vertices[topLeftVertex].y + vertices[bottomLeftVertex].y + vertices[middleVertex].y)/3 + Random.Range(-randomRange, randomRange);
    vertices[middleVertex + halfSize].y = (vertices[topLeftVertex + sizeOfSquare].y + vertices[bottomLeftVertex + sizeOfSquare].y + vertices[middleVertex].y)/3
                                        + Random.Range(-randomRange, randomRange);
    vertices[bottomLeftVertex + halfSize].y = (vertices[bottomLeftVertex + sizeOfSquare].y + vertices[bottomLeftVertex].y + vertices[middleVertex].y)/3
                                            + Random.Range(-randomRange, randomRange);

}
```

Figure 4.29: Diamond Square, Diamond and Square step.

```
void DiamondSquareAlgorithm()
{
    int iterations = (int)Mathf.Log(squares, 2);

    //Number of squares per row/column on each iteration. On the first iteration we have the whole square so the number of squares per row is the initial number.
    //Then we have 4 smaller squares, with 2 squares per row/column etc.
    int squaresPerIteration = squares;

    //Number of squares that the algorithm will perform on. On the first iteration we apply the algorithm on the whole square.
    //Then , it's divided into 4 smaller squares and the algorithm is applied on each one etc.
    int numberOfSquares = 1;

    //Step one: Initialize the height values of the four corner points.
    vertices[0].y = Random.Range(-2*maxHeight,-maxHeight);
    vertices[squares].y = Random.Range(-2 * maxHeight, -maxHeight);
    vertices[vertexCount - 1].y = Random.Range(-2 * maxHeight, -maxHeight);
    vertices[squares*(squares + 1)].y = Random.Range(-2 * maxHeight, -maxHeight);
```

Figure 4.30: Diamond Square algorithm.

```
//Diamond-square steps

for (int i = 0; i < iterations; i++)
{
    int row = 0;
    for (int j = 0; j < numberOfSquares; j++)
    {
        int column = 0;

        for (int k = 0; k < numberOfSquares; k++)
        {
            DiamondAndSquareStep(row, column, squaresPerIteration, maxHeight);
            column += squaresPerIteration;
        }

        row += squaresPerIteration;
    }
    numberOfSquares *= 2;
    squaresPerIteration /= 2;
    maxHeight *= 0.5f;
}
```

Figure 4.31: Diamond Square algorithm continued.

```
void GenerateTerrain()
{
    vertexCount = (squares + 1) * (squares + 1);
    vertices = new Vector3[vertexCount];
    UVs = new Vector2[vertexCount];
    triangles = new int[squares * squares * 6];

    Mesh mesh = new Mesh();
    GetComponent<MeshFilter>().mesh = mesh;

    GenerateVerticesAndTriangles();
    DiamondSquareAlgorithm();

    mesh.vertices = vertices;
    mesh.uv = UVs;
    mesh.triangles = triangles;

    mesh.RecalculateBounds();
    mesh.RecalculateNormals();


}
```

Figure 4.32: Diamond Square Generate Terrain function.

The Diamond Square algorithm as we can see in the figure below, gives rougher terrains with less customization options than the Perlin Noise algorithm. We reckon this kind of algorithm could be used in applications such as Flight Simulators, where the user does not interact with the terrain and instead observes it from afar.

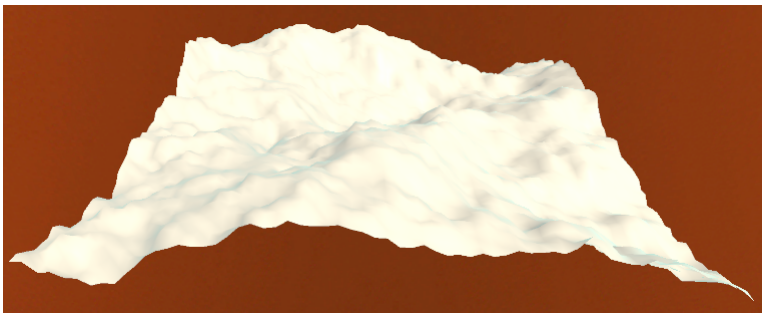

Figure 4.33: Diamond Square terrain example.

Finally, we created a custom material using Unity's Shader Graph. Shader Graph is a tool to build shaders (programs that are part of the graphics pipeline) by connecting nodes and without code. This tool allows an instant visual representation of the changes done. This custom material allows 5 different colours to render on the same mesh in different heights with their own thresholds, thus creating results like the figure below.

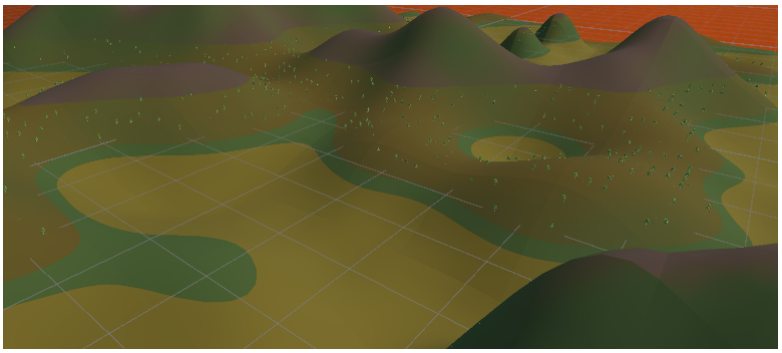

Figure 4.34: Custom material using Shader Graph example.

Chapter 5

Demo

5.1 Introduction

In this section, we will create a couple of demos to demonstrate the capabilities of this tool.

5.2 Demo 1

For the first demo we created a relatively small world with three areas. The Environment Generator Window settings can be seen below:

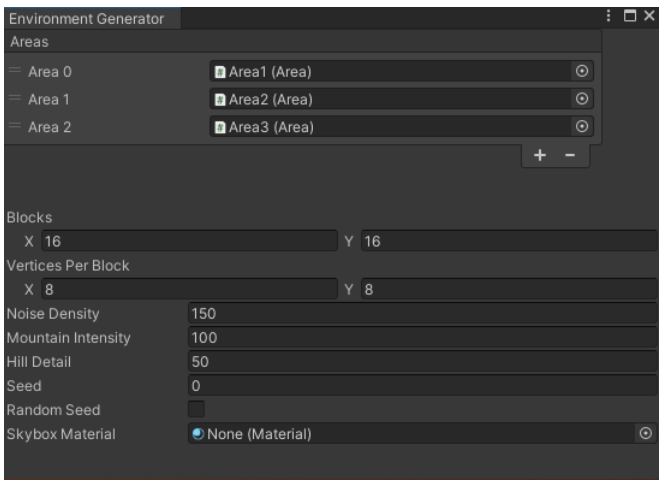

Figure 5.1: Demo 1 Settings.

The concept of this world is that half of the environment has been burned off by a fire. The first area is the healthy, green area with two vegetation zones, the third area is the burned down with one vegetation zone and the second area is empty space just with material to separate the two areas better.

Let's discuss the first area (figure 5.2). As we said it consists of two vegetation zones. The first vegetation zone (figure 5.3) is closer to the water, covering the 5-40 height range and it is meant to be a greener zone with more smaller plants since it is near the water. It consists of birch and lemon trees, and the smaller plants are grass, mushroom and flower. The second vegetation zone (figure 5.4) is meant to be more mountainous in vegetation, consisting of pines and redwood trees, bushes, mushrooms and rocks, covering the 55-150 height range.

As we can see in the figures 5.5, 5.6, regarding the first vegetation zone, we have a large amount of smaller plants spawning as we wanted. We see variety in scale and rotation of the objects and we can see that there is way more grass spawning than mushrooms for example, since the grass game object has 10 times the weight of mushrooms.

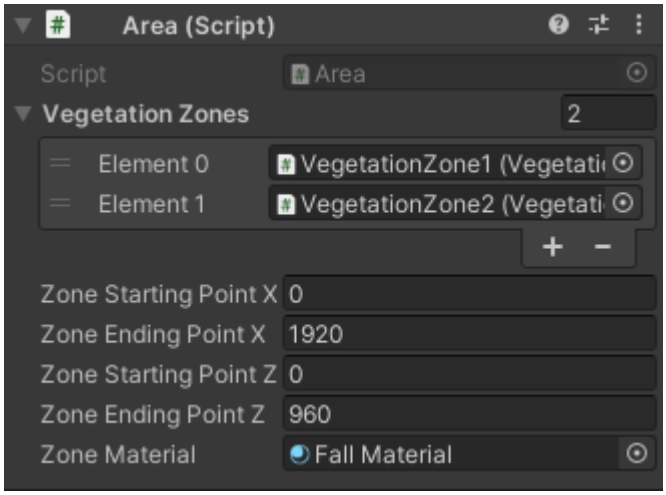

Figure 5.2: First area of Demo1.

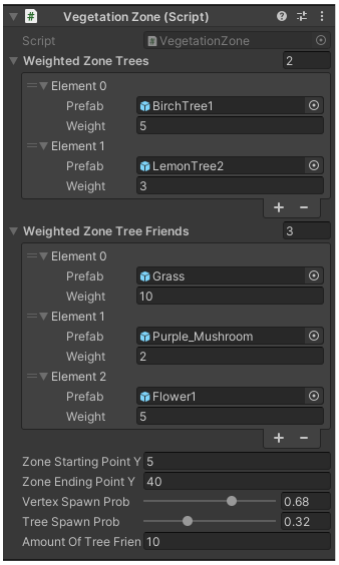

Figure 5.3: First vegetation zone of the first area of Demo 1.

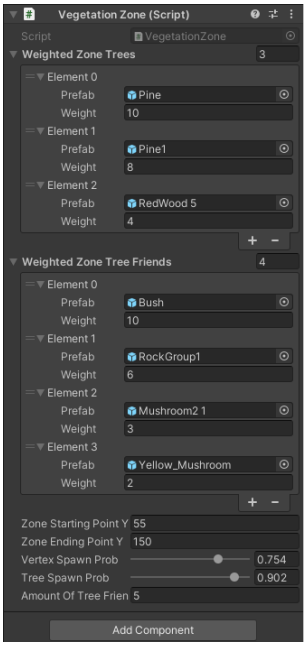

Figure 5.4: Second vegetation zone of the first area of Demo 1.

Figure 5.5: Demo 1, Area 1, Vegetation Zone 1, first example.

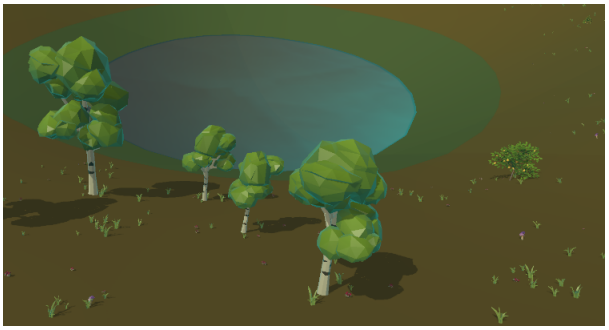

Figure 5.6: Demo 1, Area 1, Vegetation Zone 1, second example.

In the figure below we can see the results of the second vegetation zone, where we have much denser tree populations and we can again notice the weight importance with only one mushroom spawning in the bottom left corner but way more rocks and bushes.

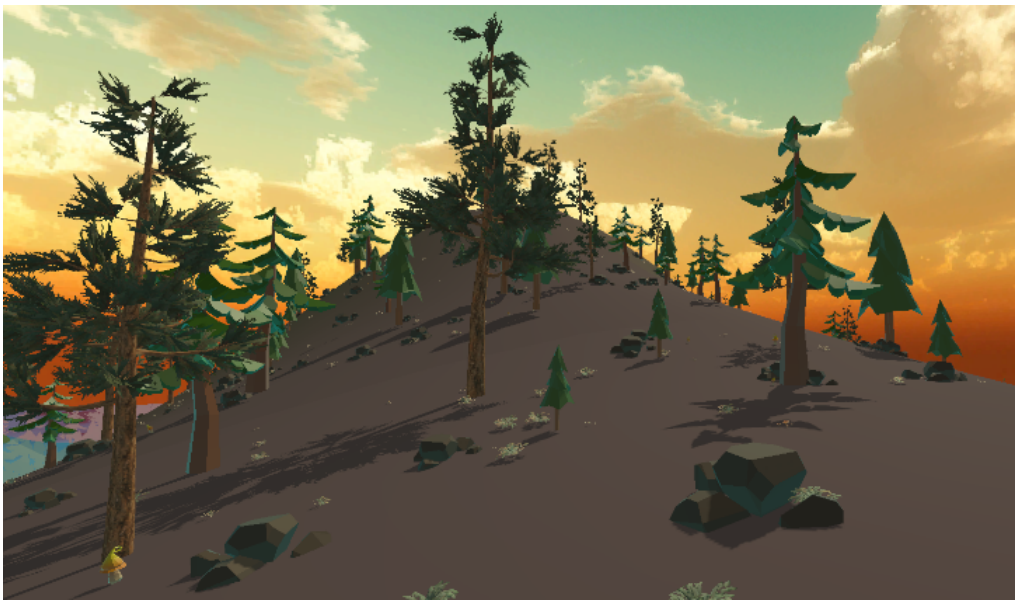

Figure 5.7: Demo 1, Area 1, Vegetation Zone 2 example.

Finally, let's see the two zones together in the figure below:

Figure 5.8: Demo 1, Area 1.

The third area covers pretty much the other half of the map. Both Area 1 and 3 have the custom material we've created, with Area's 3 material containing darker colours to give off the feeling of burnt environment. This area consists of one "vegetation" zone, consisting of leafless trees covering the whole 0-150 height range. We've also created a low poly fire VFX and attached it as "tree friend" in the Weighted Zone Tree Friends list, so that it spawns around trees giving off the impression of actual fires.

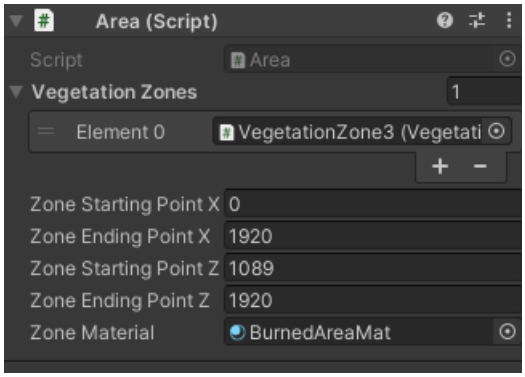

Figure 5.9: Third area of Demo 1.

Figure 5.10: Demo 1, Area 3.

Figure 5.11: Demo 1.

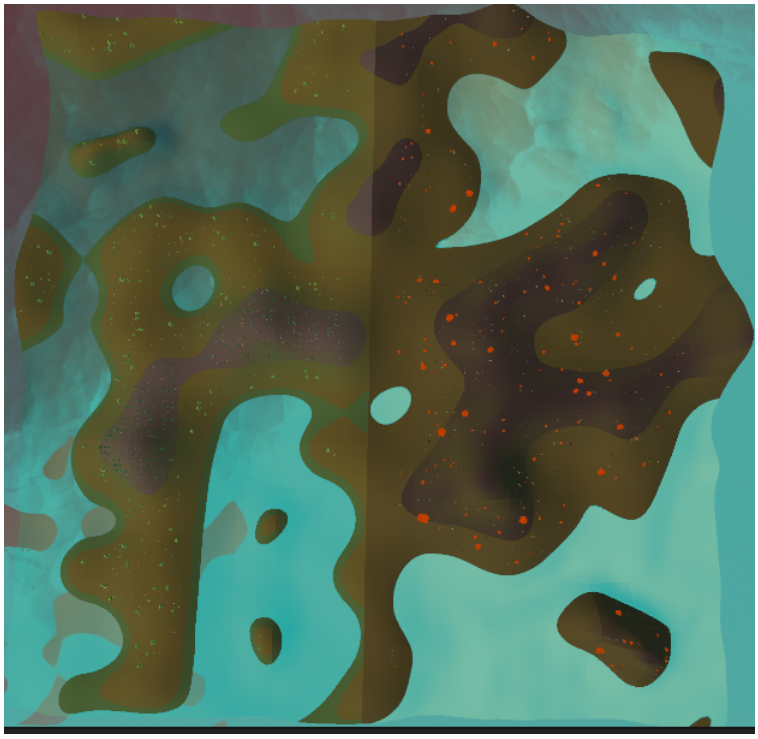

Figure 5.12: Demo 1, top down.

5.3 Demo 2

In the second demo we will create a snowy terrain which has a small transition to a greener environment. This was done with multiple different areas (specifically 8) with materials that slowly transition from one colour to the other. Of course, someone could implement a custom material with this property and only use one area, applying said material to have this effect. We can see the settings and the top down view of the map below:

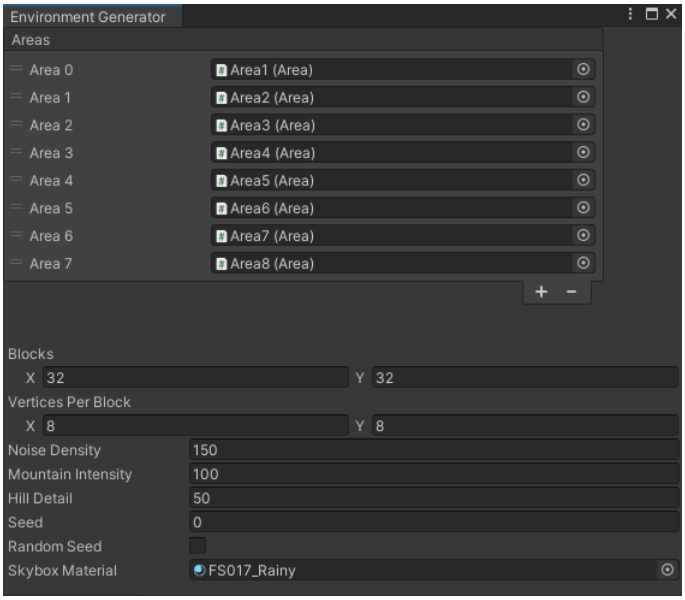

Figure 5.13: Demo 2 Environment Controller Window settings.

Figure 5.14: Demo 2 top down view.

In this environment, the left snowy part will have snowy trees and less vegetation, transitioning all the way to the right part and going greener as we move. The middle part will have a mixture of both environments. In this section, we will not show each area since there are many of them, and only show the resulting terrain. In the most snowy area, we have added some elements, hinting to human intervention. We have also added a gloomy skybox, and making the lighting darker:

Figure 5.15: First and most snowy area.

Transitioning to the middle area, closer to the snowy area, it has way more grass than trees and has a very small probability of spawning a regular, not snowed upon, tree as it can be seen in the upper left corner of the figure below.

Figure 5.16: Middle part, closest to the snowy area.

In the following part , the middle part closes to the greener area, we get a better mixture of snowy and not snowy trees, greener vegetation spawn and a mix of both trees and smaller

plants.

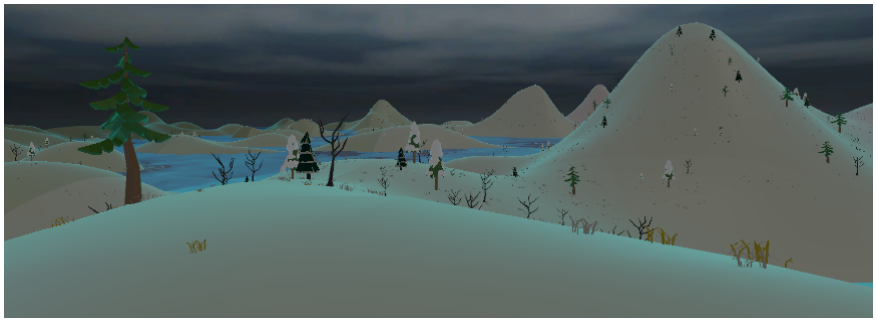

Figure 5.17: Middle part, closest to the greener area.

The final, greener part consists of denser forests with green vegetation and trees, while have a very small probability so spawn a snowed upon tree.

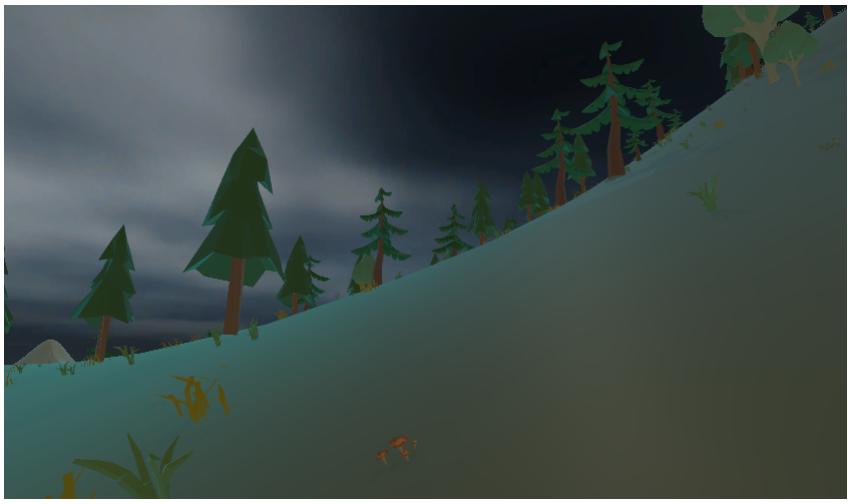

Figure 5.18: Last and greener area.

Two other good ideas for environments would be a tropical one and a dessert with an oasis. While these two demos demonstrate the capabilities of the tool, even showing that there could be some human presence in the environments, we believe that this tool can be used by more creative individuals to create much more attractive results.

Chapter 6

Conclusion, Limitations & Future Work

6.1 Summary

In this thesis, we implemented a basic tool for Procedural Terrain Generation populated with vegetation using Perlin Noise to shape the generated terrain in Unity3D. This system was inspired by games like Minecraft whose main feature is the automatic generation of different biomes, spanning across a big map.

This tool was implemented from the ground up and its purpose is to provide an easy to use, accessible and reliable way for any kind of developer to create environments, thus eliminating a tiresome process from their time. While the idea of such tools is not new, this is a practical application able to be used in a famous game engine that is used both by amateurs and professionals.

6.2 Limitations

The main limitations of this tool is the fact that it is not easy to add new laws that rule an environment that a designer might want to create. Additionally, while it could be possible to use man-made structures to create, for example, small villages, the system mainly focuses on creating purely natural environments unaffected by human hands. Finally, we should mention that, while this tool was briefly tested to see if it works like it was imagined and planned, it was not tested in a larger scale either by users or developers.

6.3 Future Work

As we mentioned, our implementation is a practical application of Noise patterns in terrain generation with the option of customizing its sub environments in a very basic form. This means that there could be many improvements done to greatly increase customization and realism of the environment.

6.3.1 Adding more features and realism in the terrain generation

The first big improvement that could be done in this system is to add more features regarding terrain shaping, such as cliffs, plateaus, e.t.c. In this tool, we sample positions from a noise map and, by doing simple mathematical operations, generate the mountains and the valleys. This main feature of this tool could be improved by adding spline points, mentioned in Chapter 2 Minecraft section. This would lead to more realistic terrain with dramatic features.

Two more features that could be added are caves and making the terrain endless. 3D noise could be used to generate intricate cave systems. In terrain generation, black parts of a noise pattern can be considered mountains and white parts valleys. In a similar way, by considering white parts as air and black as stone, we could get good looking caves. Regarding infinite terrain, it could be achieved by using rendering techniques and re-applying PCG as the player moves around the world, thus not needing to load infinite worlds.

6.3.2 Improve performance using Unity DOTS

DOTS is a new technology just recently fully released by Unity that offers developers a new way to approach game development. Up until the release of DOTS, Unity took a classic Object Oriented Programming (OOP) approach to coding. More specifically, classes represent real world things and contain variables and functions that determine their behaviour. DOTS on the other hand, present a Data Oriented Design (DOD) that focuses on how to structure data inside the memory for the system to access and process it. DOD breaks down regular objects (that are found in OOP) into components that get grouped up and stored in arrays. Finally, the system iterates across these arrays and use the data as required by the program. The framework to implemented all of the above is called Entity Component System (ECS) and is able to maximize performance at the memory and CPU level.

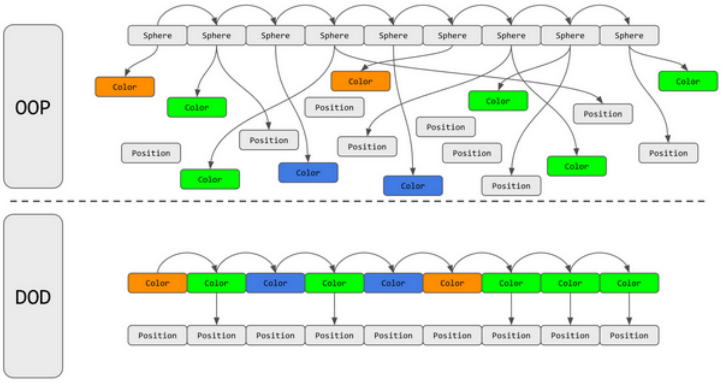

Figure 6.1: OOP vs DOD.

6.3.3 Add more rules for vegetation spawn

Another section of the tool that could be improved is the way vegetation spawns. The user can control if and what will spawn on a certain vertex by playing with the probabilities, but natural habitats are more complicated. Applying life cycle in plants, spawn being affected by slope or proximity with other plants are a few examples of rules.

6.3.4 Environment evaluation and Genetic Algorithms

In order to generate better environments, evaluation functions could be implemented giving each generated environment a certain score based on rules that we provide. By implementing such a feature, we can use an interesting family of algorithms named Genetic Algorithms (GAs) that can help our tool to produce better results. GAs simulate the ideas of natural selection and genetics. Initially, the algorithm would produce large amounts of environments randomly, evaluating them and giving them a score. This would be our initial **population**. Moving on, the environments with the best score would crossover, taking random elements from the two environments and merging them into one. We could repeat this process with the score of the environments getting better and better, but there is another option to get better performance, and that is the mutation operator. By mutating the environment we would randomly change something regarding vegetation or the terrain itself in order to maintain more diversity. When the score would not improve any more, we would pick the highest scoring environment.

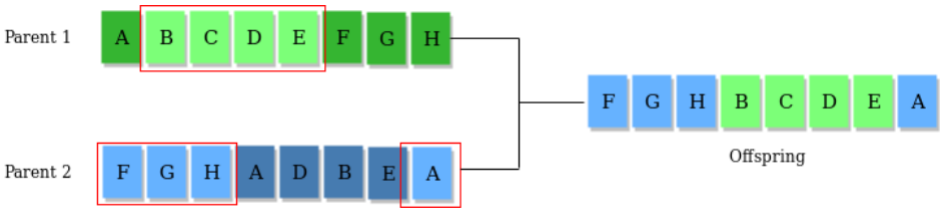

Figure 6.2: Genetic algorithm crossover example.

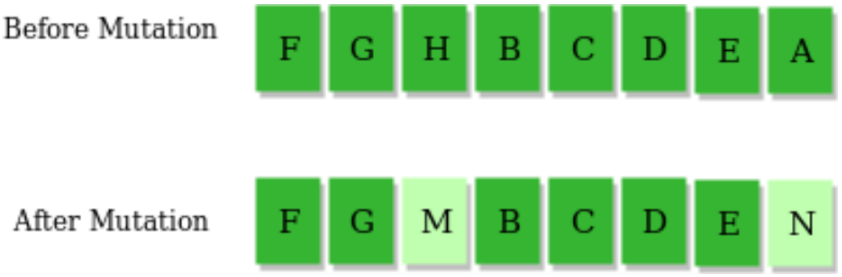

Figure 6.3: Genetic algorithm mutation example.

6.3.5 Testing results of generated environments

Finally, we've mentioned before that this tool has not been tested in a larger scale to explore its limits and fullest capabilities. To evaluate the results of this tool, users should use it applying many different ideas and parameters to generate environments.

Bibliography

[1] Shaker, Noor, Julian Togelius, and Mark J. Nelson. "Procedural content generation in games." (2016): 978-3. 6, 8

[2] Rose, Thomas J., and Anastasios G. Bakaoukas. "Algorithms and approaches for procedural terrain generation-a brief review of current techniques." 2016 8th International Conference on Games and Virtual Worlds for Serious Applications (VS-GAMES). IEEE, 2016. 1, 9, 11

[3] Archer, Travis. "Procedurally generating terrain." 44th annual midwest instruction and computing symposium, Duluth. 2011. 1, 10

[4] Smith, Gillian. "The future of procedural content generation in games." Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment. Vol. 10. No. 3. 2014. 10

[5] Hendrikx, Mark, et al. "Procedural content generation for games: A survey." ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM) 9.1 (2013): 1-22. ix, 6, 8

[6] Togelius, Julian, et al. "Procedural content generation: Goals, challenges and actionable steps." Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2013. 10

[7] Zafar, Adeel, Hasan Mujtaba, and Mirza Omer Beg. "Search-based procedural content generation for GVG-LG." Applied Soft Computing 86 (2020): 105909.

[8] Togelius, Julian, et al. "What is procedural content generation? Mario on the borderline." Proceedings of the 2nd international workshop on procedural content generation in games. 2011. 5

[9] LEE R. S. “Home videogame platforms.” The Oxford Handbook of the Digital Economy, 2012, 83–107. 7

[10] Petrillo, Fábio, et al. “What went wrong? A survey of problems in game development.” Computers in Entertainment (CIE) 7.1 (2009): 1-22. 8

[11] Andrade, António. “Game engines: a survey.” EAI Endorsed Transactions on Serious Games 2.6 (2015). 8

[12] Doran, Jonathon, and Ian Parberry. “Controlled procedural terrain generation using software agents.” IEEE Transactions on Computational Intelligence and AI in Games 2.2 (2010): 111-119. 10

[13] Bontchev, Boyan. “Modern trends in the automatic generation of content for video games.” Serdica Journal of Computing 10.2 (2016): 133-166. 10

[14] Smith, Gillian. “Understanding procedural content generation: a design-centric analysis of the role of PCG in games.” Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. 2014. 10

[15] Michael Booth. “The AI Systems of Left 4 Dead”. Valve. 2014. 7

[16] Ambinder, Mike. “Biofeedback in gameplay: How valve measures physiology to enhance gaming experience.” Game developers conference. Vol. 22. 2011. 7

[17] Lagae, Ares, et al. “State of the Art in Procedural Noise Functions.” Eurographics (State of the Art Reports) (2010): 1-19. 10