Final degree project

DEGREE IN COMPUTER SCIENCE

Faculty of Mathematics and Computer Science
Universitat de Barcelona

Procedural Content Generation for Landscapes

Author: Francisco Barrio Morra

Advisor: Anna Puig Puig
Carried out at: Faculty of Mathematics and Computer Science

Barcelona, June 20, 2021

Abstract

Procedural Content Generation is the act of creating content by using a process instead of predefining it. This project proposes generating different terrain types in a procedural way, while letting a player move around it and explore it freely. This means that the terrain will be generated in real time as the player moves, and how it is generated will be affected by the player's actions, and how they explore.

The generation of this terrain has regions with different characteristics which make them have a specific appearance, with the transitions between them being quite seamless. The objective was to generate a terrain that was as close as possible to real terrain surfaces on Earth.

The developed application lets explore a wide range of configurations regarding the PCG, getting very promising results in terms of player's enjoyment.

Resumen

La Generación de Contenido Procedural es la creación de contenido mediante procesos en vez de usar contenidos predefinidos. El objetivo de este proyecto es generar distintos tipos de terrenos de una manera procedural, todo mientras se le deja al jugador moverse a través de él y explorarlo libremente. Esto supone que el terreno será generado en tiempo real mientras el jugador se mueve, y el cómo será generado se verá afectado por las acciones del jugador y por cómo explora.

La generación de este terreno contará con regiones con distintas características cada una, que harán que tengan una apariencia específica, y haciendo de las transiciones entre las regiones lo más suaves posibles. El objetivo es generar un terreno que se asemeje lo máximo posible a un terreno real del planeta Tierra.

Acknowledgements

I would like to thank my family, especially my parents for supporting me financially and emotionally throughout my entire life.

I would also like to thank my tutor, Anna Puig Puig, for guiding me helping me immensely during the project.

Finally, I would like to thank my friends Aarón, Alejandro, Arnau, John, Manuel and Pau, who have entertained me when I needed it most.

Contents

1 Introduction **1**

2 Background **5**

2.1 Problem definition 8

2.1.1 Context 8

2.1.2 Terrain 9

2.1.3 Interconnection 9

2.1.4 Materials and textures 10

2.1.5 Interaction and real-time creation 10

2.1.6 Adaptability 10

3 Application design **11**

3.1 Cave-like structures 11

3.1.1 Cellular Automata approach (2D-based) 12

3.1.2 Mesh generation of cave-like structures 16

3.1.3 CA for 3-dimensional caves 24

3.2 Terrain generation 28

3.2.1 Noise 28

3.2.2 Surface generation 38

3.3 Biomes 58

3.4 Texturing 60

3.4.1 Biome material shader 60

3.4.2 Sewing material shaders 64

3.5 PCG adaptability . . . 66

4 Software Design 69
4.1 System architecture . . . 69
4.2 Components and GameObjects . . . 70
4.3 Project structure . . . 72

5 Simulations and Results 77
5.1 Simulations . . . 77
5.2 Results . . . 83

6 Conclusions and Future work 87
6.1 Conclusions . . . 87
6.2 Future Work . . . 89

A Technical manual 91
A.1 Software versions . . . 91
A.2 How to install . . . 91
A.3 How to execute . . . 91

Bibliography 95

Chapter 1

Introduction

We are all familiar with videogames. Programs where we are set in a world and must accomplish an objective, both set by the creator. But, what happens when the creator wants the game to look or feel different every time we play it, without having to define a list of possible configurations? This is where Procedural Content Generation comes into play...

Procedural Content Generation (PCG for short) [19] is all the content that is created in an automatic way. This means that any type of content you want to generate - meshes, objects, levels, etc. - is created by different algorithms which give a different output based on a number of parameters or input, instead of being predefined by the creator. An example of this is a game where we generate a maze that the player has to find the exit to. Instead of defining the maze ourselves and loading it when the player starts the game, we generate a different maze every time the player plays the game. This is not strictly limited to videogames, though, as we could create a program that generates a painting in a specific author's style out of a picture of a landscape.

Context

This project was developed in the context of a Computer Science degree, and thus was influenced by a lot of the matter learned in it. A game will be made using the Unity 2019.2.17f1 game engine, and even though a game engine was never used in the degree, contents from the following subjects in particular have helped or been needed:

- Programming: object-oriented paradigm is widely spread in game development.

- Graphics: learning about what meshes are and how they are constructed beforehand has helped immensely with the project, as well as knowing how GPUs work with textures, for instance.

- Parallel Programming: in order to optimize the performance of the game, the use of multithreading was needed.

Motivation

For this project, we will be focusing on world generation, particularly, the generation of a landscape. This is an aspect of game development that's becoming more and more important as time progresses, since games tend to be of a bigger and bigger scale. Lots of games want to have procedurally generated worlds, mainly because this adds the possiblity of making them endless, and add the possiblity to be replayed over and over, as well as giving a sense of exploring the world, as you never know what you can find there. This is the future for a big part of game development, and a very interesting and important topic to research.

It is also a technological challenge to be able to generate a reactive system that is endless, since, not only is it important to guarantee a number of requirements from the terrain based on geometry, topology and textures, but also giving the possibility of managing the time and memory resources efficiently, so that it doesn't collapse after a certain amount of playtime, all while giving interactive gameplay.

Especially when mixed with other advanced technologies such as Artificial Intelligence, we think PCG will be the future not only of videogames, but the whole entertaining industry. This includes possible profitable branches such as having virtual worlds with procedurally generated content for applications like e-commerce, where clothes and apparel could be created as PCG, online education, where creating virtual worlds could help students - especially visual learners - understand some concepts better, and so on.

Main goal

The main goal of this project is to generate a landscape using procedural world generation techniques. It must be somewhat realistic, that is, resemble in some

way to reality. It must also be able to be explored by a player the way they want to, so it must be generated in real time.

Additionally, we also wanted the world to be able to adapt to the way the player thinks or plays. So, we will introduce aspects of it that allow the world to change depending on what the player does.

In essence, we want to create a small scenario where the player has the sensation of exploring a realistic endless world that adapts to his interests.

Specific goals

The main goal will be divided into the following, more specific sub-goals:

- **Analyzing the most common PCG techniques.** This means studying some techniques commonly used for generating different types of content.
- **Designing a game environment to include different PCG techniques.** Creating an environment that allows the techniques we will implement to be interactive.
- **Analysis, design and development of cave-like environments using PCG techniques.** First step in understanding how some specific terrains are commonly generated.
- **Analysis, design and development of landscape using PCG techniques.** An introduction to how relief terrain is generated using noise, and how to tweak different parameters to obtain more desirable results.
- **Analysis, design and development of chunk generation techniques and its parallelization.** A way of generating terrain in a way that is endless and in real-time.
- **Analysis, design and development of biome-based distribution of chunks.** How to distribute the different terrain chunks to obtain realistic terrains at a large scale.
- **Design of player adaptability.** Finally, a way for the world to adapt to what and how the player likes to explore.
- **Validation of the final deployment** in terms of tests and simulations.

Planning

The project was planned week by week, initially as seen in Figure 1.1.

Goal	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Analyzing the most common PCG techniques.	■	■	■																	
Designing a game environment to include different PCG techniques			■	■																
Analysis, design and development of cave-like environments using PCG					■	■	■	■												
Analysis, design and development of chunk generation techniques and its parallelization									■	■	■	■	■	■						
Analysis, design and development of biome-based distribution of chunks															■	■	■	■		
Design of player adaptability																			■	
Validation of the final deployment																				■

Figure 1.1: Initial planning of the project, week by week, grouped by task.

Chapter 2

Background

PCG has been an interesting and researched topic since a long time ago. The first appearance it had on videogames (which is what we will focus on for this project) was around 1978 when a game called *Beneath Apple Manor* generated dungeons procedurally [1]. It could define rooms, hallways, treasures and enemies the player could fight, all this on the fly.

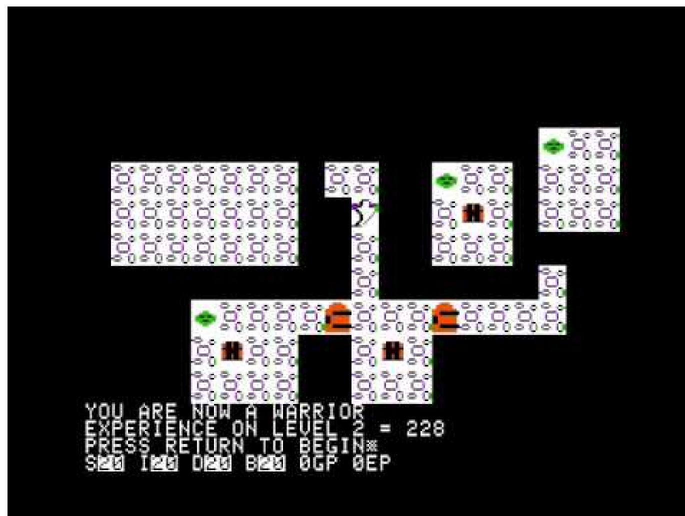

Figure 2.1: Example of a dungeon generated by the game *Beneath Apple Manor*

Back in the time, memory was a problem. This sparked a need for content to be generated on runtime, instead of being stored in memory, and this allowed for games to be much lighter. Then, the era of CDs and DVDs came, and with them, more memory was available, and the need to generate content in runtime

was somewhat discarded, instead opting for a higher quality and control over the content. Videogames also started becoming bigger and bigger and started being seen as a way to tell a story, much like a movie or book do, except the latter lacked the interaction aspect the former contributed. This meant that PCG wasn't really used, since the creators wanted to tell a story in a certain way. As they got bigger, they also thrived for a higher quality, which was difficult to achieve using PCG.

Later on, the roguelike genre [10], which usually consists in advancing through a dungeon, defeating enemies or collecting treasures, started getting more and more popular, and they were usually fast-paced and short games, which focused on replaying the game to achieve progress, so they needed PCG in order to stay entertaining and fresh. And so, PCG started becoming a more widespread topic.

Figure 2.2: The Binding of Isaac, a roguelike game where every level is composed by a number of rooms, and the player must clear each room of enemies to progress to the next. On the right, we can see the room disposition.

As we can quickly deduce, this is a really powerful tool, mainly because it allows for a lot of variety in the generated content, varying - sometimes drastically - between iterations. This gives the player the feeling that they're playing a different game every time they start a new game - to some extent, since the core mechanics will still be the same - which helps with fighting boredom.

The most basic approach we can think of about creating content on the fly, is to have a random result, independent of anything else. In our previous example, this would be equivalent to making a completely random maze, where we divide it in tiles, and every tile has a probability of either being a wall or not. If we think about it, this isn't a very good way to generate a maze, since, we could very easily end up with an unsolvable maze with no exit, a maze where all tiles are walls, or the opposite. Using this method, these last two outputs are unlikely, but possible. Another simple example would be trying to generate random names for

a character, assigning completely random letters. This would give us names that are really hard to pronounce, and completely absurd ones, such as "a". Knowing this, we can understand why procedurally generated content is usually created following a set of rules and constraints, which we can define to tweak the final result we want to obtain. This way, we can define which outputs we are able to obtain and which we are not. Obviously, the more rules and constraints we define, the more similar all our content will be, but without no constraints, as we saw before, it is practically impossible to generate good content. So, it is all about finding balance between our creative freedom and our integrity and consistency. An example of this on our maze would be using a constraint that states that all the outer tiles except one (which will be the exit) have to be walls. This way, we ensure that the maze will at least be surrounded by walls, and have an exit.

This is much faster and can give much more content than we would be able to define ourselves. This has a drawback, though, which is that we sacrifice some quality in favor of quantity. Since we don't define the content directly, our rules will have to be really good if we want to obtain results similar to those we would think of ourselves instead of using algorithms. But once we have defined those rules just as we want, we will obtain lots of good results, so we have to trade spending time defining the rules in favor of having a practically infinite content span, which, depending on the content we want to create, can be extremely worth it.

Practically anything can be procedurally generated, from worlds, to characters, to text, to textures. Combining all these aspects in a game can yield an extreme variation between iterations and result in a sensation of playing completely different games, thus giving our games or puzzles a very high potential to be replayed. PCG is a very widespread topic with a lot of branches and a lot of interest put into. Thus, exploring its intricacies is a very tempting offer.

The problem with PCG is that it is hard to achieve realistic results, and it usually takes a lot of time tweaking parameters. This is especially true for a project of this nature, where realism is on the spotlight and its main attraction. Although, not all content generated procedurally must be realistic or resemble any model at all whatsoever.

One possibility is distributing the biomes depending on real physical parameters such as temperature and precipitations, just like they are on Earth [2]. We thought this was interesting since it mimics the way they are distributed on Earth pretty closely, although we thought this might be too realistic for an infinite world, since, on Earth, temperatures and precipitations are based on other factors such as the distance from the equator, the distance from a mass of water, and such.

Thus, we thought this might not be the perfect design choice for our interests, since we had no way of realistically simulating these external factors, so we stuck to a simpler approach.

Games like the 2016 title *No Man's Sky* [5] use this biome distribution to generate millions of different planets with different characteristics such as extreme cold or hot, toxic environments, or other factors. This way, they can generate procedural planets that are quite distinct, as well as generating different flora and fauna which is also generated for each planet. We will be inspiring our work in this idea.

Figure 2.3: A planet generated procedurally by the game *No Man's Sky*. We can see the different biomes, such as a forest on the left and a beach on the right.

2.1 Problem definition

In this section we will introduce the main topics related to the PCG addressed in this project. The problem can be divided in several smaller problems that will need to be addressed.

2.1.1 Context

Mainly, we want to build a scenario which consists of a landscape with different zones, which can be softer or more abrupt, called biomes, such as hills, plains, or mountains. Examples of these biomes are seen in Figure 2.4.

Mountains

Hills

Plains

Figure 2.4: Examples of mountains, hills and plains biomes we will implement in this project.

2.1.2 Terrain

We want to construct a scene in a 3-dimensional virtual world, formed by triangle-based meshes that fulfill certain requirements, such as not ditinguishing between different toplogies and topographies when it comes to how they're constructed, or not giving unrealistic results such as *floating islands*, which we wouldn't see on Earth. It is not such a trivial task to generate a mesh out of vertices, since several processes are involved, such as placing the vertices correctly in the world space, and defining what the triangles of the mesh will look like. All this while making the meshes have a variable size defined by the user.

2.1.3 Interconnection

As we said, it is not trivial to define a mesh correctly, and even less so to make multiple meshes connect together seamlessly. The generation will be split into chunks, where each chunk will be generated following certain parameters typical of their respective biomes, and we will need to guarantee the connection between two chunks that belong to different biomes is seamless, and the mesh has no gaps.

2.1.4 Materials and textures

Not only do the meshes themselves and the transitions between them need to appear seamless, in the real world terrain has color and texture, and to model that, especially while keeping everything that was previously commented together is not a trivial task either. So, in this project, we will investigate how to manage different textures and materials to model the biomes we will be implementing.

2.1.5 Interaction and real-time creation

Creating a terrain in our world not only is not a trivial task, it is also a slow one. Using different optimization techniques, it is important to optimize the generation of said terrain in order for it to be quick enough to be generated in runtime, adapting to where the player wants to go.

The need to create different chunks of the terrain also poses a challenge regarding this real-time generation, optimizing the time and memory usage as well as possible.

2.1.6 Adaptability

Giving the player the ability to impact what is generated is, like everything else, not trivial. Since the player is generally unpredictable and could try to trick the system on purpose, it is difficult to account for this.

It is also a challenge to decide how to let the player affect the content generation in a way that makes sense to them and doesn't negatively affect the experience. For instance, making the terrain generate more chunks of a specific biome the player likes to explore, or doing the opposite and forcing the player to "chase" the biomes they like if they want to keep seeing them.

Chapter 3

Application design

In this chapter, we will introduce some concepts we will use to generate our world, as well as explaining the strategies we will propose to create it in more detail.

Normally, techniques that generate content, especially for terrain, use either iterative processes, where a function is repeated over the terrain, changing it over every iteration, and stopping when we get a result that is close enough to the desired one, or pseudo-random functions, such as Perlin or Simplex noise. We will get into the former at a later point in the project.

3.1 Cave-like structures

In this section, we will talk more in depth about a specific terrain structures that can be generated using PCG, as an introduction to it: caves, and we will propose some techniques to implement them.

A cave or cavern is a natural void in the ground, specifically a space large enough for a human to enter. Caves often form by the weathering of rock and often extend deep underground. The word cave can also refer to much smaller openings such as sea caves, rock shelters, and grottos, though strictly speaking a cave is exogene, meaning it is deeper than its opening is wide, and a rock shelter is endogene. [15]

Caves are characterized by being closed off by rock walls, except for one or a few narrow openings to the exterior, usually with stalactites hanging from the ceiling as a result of sediments carried by underground water currents being deposited as the water drips down to the floor, and stalagmites, formed by the same

water drops depositing their sediment once they reach the floor instead of doing it before falling from the ceiling (see an example in Figure 3.1.

Figure 3.1: Example of a cave.

3.1.1 Cellular Automata approach (2D-based)

In video games, caves are usually represented as maze-like structures with poor lighting and rocky walls. So, as a first contact with PCG, we generate a cave.using a technique based on a *cellular automaton*.

2D cellular automata [22] are a discrete model of computation, consisting on a grid of units called *cells*. These cells are in one of a finite number of well-defined states. A cellular automata is updated on every iteration, and on every update, each cell will change its state depending on its own state and its *neighbours*'. Cells' neighbours are simply the cells that every cell will look at when deciding whether to change states, apart from itself. This neighbourhood is usually defined as the 8 adjacent cells - in the case of a square grid -, although this number can be bigger. Basically, every cell will look at every other cell that is at less than n cells away - this n is a parameter we can tweak. Thus, when n equals 1, every cell will look at its 8 directly adjacent cells. When n equals 2, every cell will look at its 8 directly adjacent cells, as well as the 8 directly adjacent cells of those.

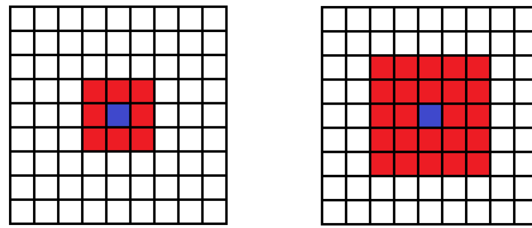

Figure 3.2: We can see the blue cell's neighbours in red, with n = 1 on the left, and n = 2 on the right.

Normally, the cells of a cellular automata have 2 states - dead and alive. This is useful because this way, we are able to easily represent a cellular automaton, assigning a 0 for the dead state and a 1 for the alive state. This isn't necessarily true anymore, since we can use lots of states using other data types such as integers. Although, it is worth noting that the complexity scales up pretty fast with the number of states we have, since we will have a bigger number of transitions from state to state. Thus, keeping our number of states as low as possible is always going to help us.

Cellular automata (which we'll call CA from now on) is really useful for creating caves, since we can define a state for a cell being a wall, and a state for a cell being not a wall. Then, we use the following algorithm in order to generate caves (although, there are a lot of algorithms, including those which don't use CAs, that achieve this. Also explained in [8]).

First, all cells will start at a given state, depending on how we define it. For example, all border cells can start at 1 (wall state), while the other cells start at 0 (non-wall state). This is optional, and will guarantee that our cave is completely closed off. Alternatively, we could have all our non-border cells to start at a random state, in order to have random openings that would lead to the outside of the cave.

Figure 3.3: Example of what a 100x100 CA would start like if we have a border size of 3, and have our cells start at a random state. Each pixel of the image represents a cell, where black and white pixels are walls and non-wall cells, respectively. On the left, each cell has a 30% chance of being a wall, and on the right, each cell has a 70% chance.

Second, after generating the starting CA, we then proceed to repeat an iterative process, where we'll change the state of each cell according to a number of parameters.

- *Smoothing.* The number of iterations our CA will go through, updating each cell. We call it smoothing because, as we will see now, the higher it is, the smoother our caves will be, since, as we will also see, each iteration will get rid of isolated walls.
- *Number of neighbours.* For now, we will take only the immediate neighbours - if we remember what we talked about before, this means an n equal to 1.
- *The laws to follow to change the state.* The rules we are about to describe are arbitrary and designed for our purpose specifically, although they may be changed in order to obtain different results. A famous CA designed by John Conway, called *Game Of Life* [7] has a different set of rules that try to mimic live cells, and their reproduction and death.

As an example using concrete values for these parameters, we could do, for each cell:

- If the cell is a wall, and has more than 4 neighbouring walls, it will stay as a wall.
- If the cell is not a wall, and has more than 5 neighbouring walls, it will become a wall.

- Otherwise, the cell will stay as or become a non-wall cell, depending on whether it wasn't or was a wall cell, respectively.

It should be noted that the 4 and 5 mentioned here are threshold parameters that we can tweak in order to alter the results.

Knowing the iterative process, the parameters and the laws for changing states, we can deduce the effect they will have on the grid if we think about it. The non-wall cells that are surrounded by a good amount of walls (in this case, 5) will turn into walls, and the wall cells that are also surrounded by a good amount of walls (in this case, 4) will stay as walls. This means that walls will only be kept or formed around other walls, so wall groups or clusters will begin to form (Figure 3.4 shows the evolution of the iterative process).

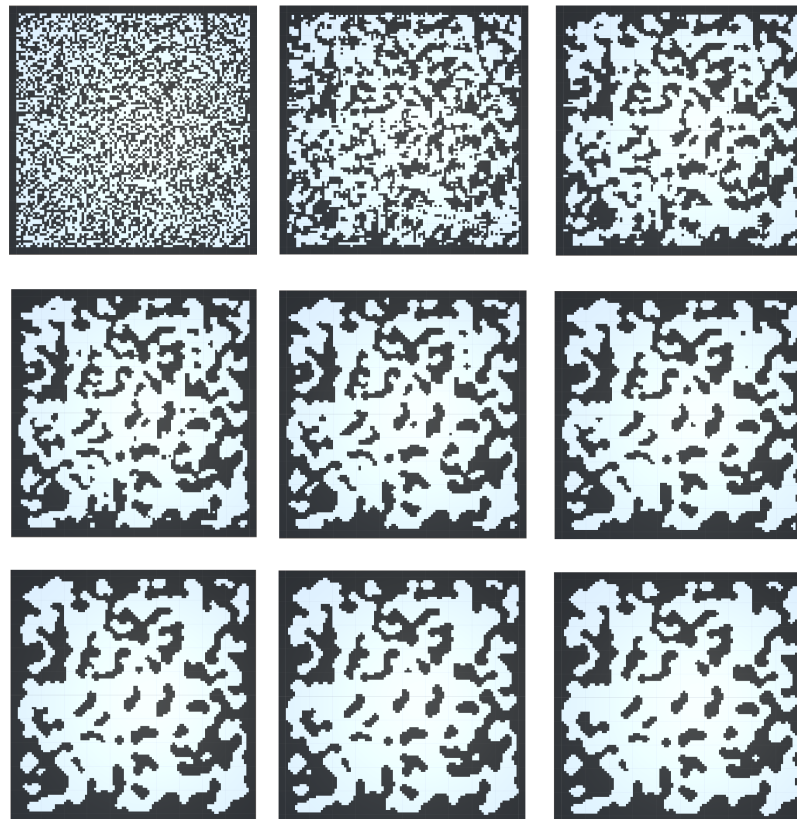

Figure 3.4: In this image, we can see the result of the iterative process after each iteration (starting on the top left corner, then going to the right and down), up to 8. As we can see, the different wall cells start to appear and disappear, forming clusters and leaving empty space between the clusters. Each image has a 100x100 resolution, where each pixel is a tile.

After all, we could an algorithm to guarantee the connection between all empty spaces of the cave. The algorithm basically consists on finding every empty space's area using a filling algorithm, after which we find the closest point between every pair of empty spaces, and connecting them using a hallway. This is an interesting algorithm, but we determined that it wasn't useful for our project, since, if at any point we decided to include a way of modifying the cave, it would be interesting to have isolated empty spaces in order to hide treasures, or just because realistically, caves do sometimes have these spaces in them.

It is worth noting that this algorithm can also be applied on 3 dimensions (and generally, on any number of dimensions), although it is quite difficult to represent this on 2 dimensions, and we will talk about it more in depth on the next section.

3.1.2 Mesh generation of cave-like structures

In 3D computer graphics, objects are represented in a virtual 3-dimensional world, which has 3 coordinates - x, y, and z - that represent its 3 axis - width, height, and depth, respectively. In order to have our map grid represented in a 3-dimensional world, we will need to build a mesh in our 3-dimensional world, which will store different heights in order to represent the walls.

A **mesh** or **polygon mesh** is a collection of vertices, edges and faces that define a polygonal or polyhedral object [3]. These meshes are, as we said, composed of vertices - 3-dimensional vectors that represent a point in space -, edges - combinations of vertices that represent the connections between them -, and faces - combinations of edges. These meshes will be a very important part in our project, since they're what the Graphics Processing Unit (GPU) will use in order to project our 3-dimensional objects - in our case, terrains - to a 2-dimensional raster, a process that takes a 3D scene or part of a 3D world and transforms it to an image needed to represent it on a screen, since the screens we use are flat and can only show 2 dimensions. This process is shown in more detail in Figure 3.5.

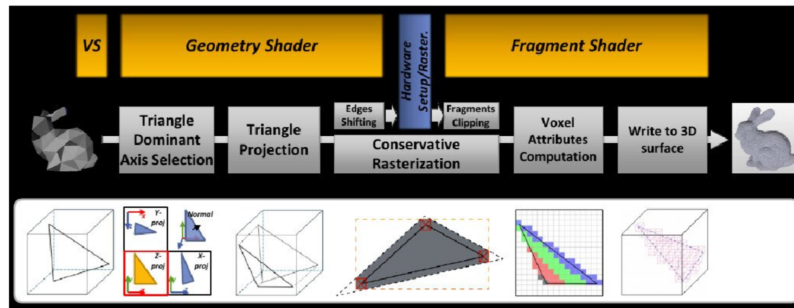

Figure 3.5: How the rasterization process works on a GPU. We won't get into too much detail about how this works, we only need to know it takes a 3D scene and projects it into a 2D image [13].

In order to build our mesh to represent the caves we built previously, we will need to, first, define its vertices. This process is relatively simple. For the x and z parameters, which are the coordinates of our *ground* plane, will just need to be set to the center of our mesh, plus an offset that will depend on their position on our cave array. We will explain this part on more detail later on, as it will make more sense in context. All that needs to be done for the y parameter of our mesh vertices - the height in our 3D world - is to assign it to a certain wall height parameter. Thus, for now, all the walls will have the same height. This will give a feeling that the cave's ceiling has been cut along the x-z plane, and allows us to see inside it. See Figure 3.12.

To show an example, let's say we will create a mesh to represent the cave we built previously in Figure 3.4, and we will center it on the coordinate (0, 0, 0) of our virtual 3D world. Let's also say that our cave will be 100x100 tiles - that is, 100 tiles wide and 100 tiles deep. Now, in order to understand a little bit better how our mesh is exactly built, we need to know a little more about how our cave was built.

Computationally, we will use an array to represent our grid array, and as we mentioned before, our grid contains a number - which could be either 1 or 0, depending on whether a position is a wall or not -, so our array will hold this number itself. Since it is a 2-dimensional grid, we will use what's called a 2-dimensional array.

This is important to explain because, in order to build our mesh, we will have to define its vertices giving them a position, or more commonly called, an **index**. This index has to be unique and identifies every vertex. This is used to, later on, construct the triangles which are what the GPU will know how to represent on the screen.

The way the cave array - which we will call **heightmap** from now on - is built is simple, we start at the lower left hand corner, work our way up, and once we reach the limit on the y axis, we go back to the bottom, except now we are a column to the right. We then assign the step for each iteration. We can take advantage of this iterative process to give every vertex in our mesh an index. So, our vertex indices will take the step of the iteration its counterpart in the heightmap was created on. After that, our indices will look like in Figure 3.6.

4	9	14	19	24
3	8	13	18	23
2	7	12	17	22
1	6	11	16	21
0	5	10	15	20

Figure 3.6: Example of how our vertex indices would be distributed if our grid was 5x5.

As we said, this distribution is arbitrary (it is possible to distribute them the other way around, starting with 0 on the upper left corner and working your way down and to the right, or any other combination), although we chose this one because it was the most intuitive by itself, and also when it comes to building the triangles the mesh will be composed of. For this, we will join three adjacent vertices as we show in Figure 3.7.

4	9	14	19	24
3	8	13	18	23
2	7	12	17	22
1	6	11	16	21
0	5	10	15	20

Figure 3.7: Example of how we will join our vertices via triangles. Each triangle in red represents one of the triangles our mesh will have.

And thus, our list of vertices (which are, in essence, 3-dimensional vectors) will look something like what's shown in Figure 3.8.

Vertex index	x	y	z
0	0	h	0
1	0	0	0.3
2	0	0	0.6
3	0	0	0.9
4	0	h	1.2
5	0	0	1.5
6	0	h	1.8

Figure 3.8: Example of how our list of vertices will look like. Note that we use an array to store it, and the vertex index is not stored in it, it is implicit. It is also important to note that the h here is a height parameter. Thus, all walls will have a height of either 0 or h. See Figure 3.12.

It should be noted that the triangles of the mesh are the most important part when it comes to visualizing it, because they are what the GPU will calculate the raster of, determining if a triangle occupies a certain pixel or not, and giving that pixel the color of the triangle (taking into account its material, lighting and other factors such as texture, but we'll go into those later into the project).

A triangle, as we know, is a polygon with 3 vertices, and thus, it is represented via a data structure containing 3 integer numbers, which are the indices of the 3 vertices that compose it. The order is important here, though, since there is a trick GPUs use in order to decide whether they need to represent a triangle or not. This trick takes advantage of the fact that, even for those triangles that would be on the camera scope, not all of them are visible. For example, let's imagine we are holding a football on our hand. A sphere such as a football is composed of lots of little triangles, since in computer graphics, a sphere is actually just a polyhedron with a lot of faces, the more faces it has, the more it resembles a sphere, but the more computationally expensive it is to store and represent visually. If we look at this football, we will see that the back of it isn't visible to us, since our vision is actually 2-dimensional (we see a flat image), and, based on lighting, shadows, and

the fact that we have 2 different perspectives (our 2 eyes), our brain then lets us interpret the depth of objects. But we cannot really see the 3 dimensions of objects, since the way we see is thanks to light bouncing off of objects and reaching our eyes, but light doesn't go through opaque objects, so the light that bounces off of the back of an object in respect to us doesn't reach us because it is blocked by the object itself.

Thus, we can take advantage of this trick to decide whether or not we want to draw a triangle on screen based on whether this triangle is at the back or at the front of our object in respect to our perspective (in this case, the camera). With our approach, we define a *direction* for all the triangles, which will depend on the order we define its vertices in. This direction can be either clockwise and counter-clockwise. Then, when the GPU decides which triangles to draw and which not to, it will only draw those triangles that have a clockwise direction with respect to the camera (it is important to note that this is arbitrary, and it was chosen like this as a convention for the engine we used for this project (Unity), while other graphics APIs such as OpenGL have counter-clockwise as the default winding direction, thus, this process would work the other way around). This process is called back-face culling [12]. We can see this more clearly in Figure 3.9.

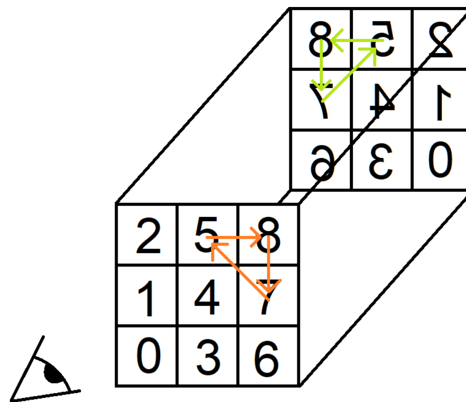

Figure 3.9: Example of how the process of back-face culling works. Let's imagine we have defined a prism with 2 opposite faces that are equal, defined with a 3x3 array with these vertex indices. As we can appreciate, if we define a triangle with the 3 indices (7, 5, 8) - in that order, or any representation that maintains that relative order, such as (5, 8, 7) -, when the face is facing the camera, the direction of the triangle will be clockwise (shown in orange), whereas in the face that is not seen by the camera, the same triangle now has a counter-clockwise direction (shown in green), and thus, it won't be shown.

Thus, if we were to look at this object from the front, it would look something like in Figure 3.10.

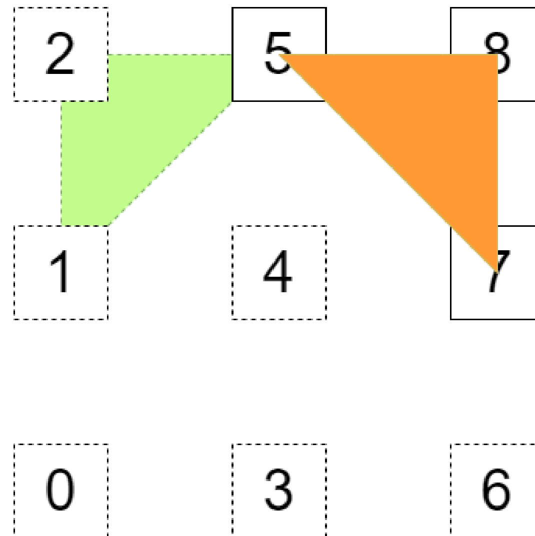

Figure 3.10: Example of how the object seen in Figure 3.9 would look like if seen from the front, assuming the only triangles defined for it are the orange and green triangles shown in that same figure. Note that only those elements drawn with solid lines would be shown on the screen, the ones drawn with dotted lines would not.

Notice how the vertex information itself doesn't matter for the construction of the triangles and thus our way of constructing a triangle will not depend on the position of its vertices. Now, we can see how the mesh for our cave will be built.

We will iterate our array from the first index up to the end. Then, for each iteration, we will build 2 triangles, one formed by the vertex itself and 2 other vertices, and the other one formed by the other 3 vertices that are part of the same square as the original vertex. We will see an example of this now. This is needed because the indices of the last row and last column can't form any triangles in their iteration since they don't have any vertices above them, and instead will be part of the triangles defined by the previous iteration.

If we look at how vertices are defined in Figure 3.9, we would start our iterations at the index 0, and we would build 2 triangles:

- The triangle formed by the vertices (0, 1, 3), which we'll call just by the vertices that form it.

- (3, 1, 4)

Then, on the next iteration, we would be on the index 1, and we would build the following triangles:

- (1, 2, 4)
- (4, 2, 5)

These triangles would look like it's shown in Figure 3.11.

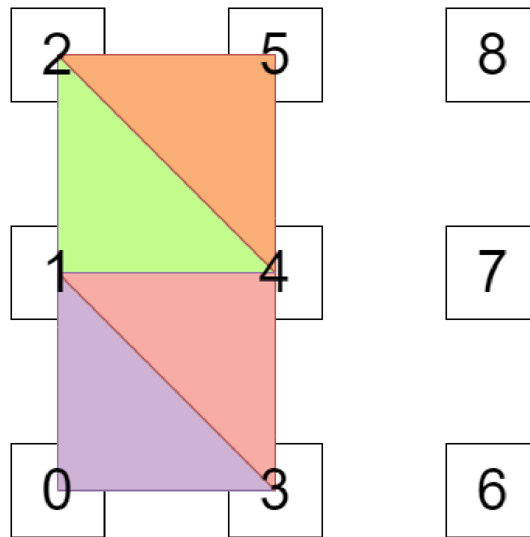

Figure 3.11: Example of how the triangles defined in the previous example, triangles (0, 1, 3), in purple, (3, 1, 4), in red, (1, 2, 4), in green, and (4, 2, 5), in orange, would look like. As we see, on iteration 0, we use that index, and the 3 other vertices that are part of the same square, which are 1, 3 and 4, to construct the triangles.

Actually, we would repeat this process for every vertex up until the $(M-1)*(N-1)$th one if our array is of MxN dimensions - that is, a width of M and a height of N. This is due to the fact that, as we can see, if we build the triangles this way, the vertices on the last row or column don't have to build any, since they will already be part of the triangles defined by the previous iterations. This means that the vertices 2, 5, 6, 7 and 8 (and thus, those iterations) of Figure 3.11 won't create any triangles. For instance, the vertex 2 will already be part of the triangle created by the index 1, and the vertex 5 will already be part of the triangles created by the index 1 and 4. The same way, the vertex 6 will already be part of the triangle

created by the index 3, and the vertex 7 will already be part of the triangles created by the indexes 3 and 4. Finally, vertex 8 will already be part of the triangle created by the index 4.

As we can see, the number of triangles we will have on a mesh will be equal to

$$(M-1)*(N-1)*2 = (MN-M-N-1)*2 = 2MN-2M-2N-2$$

since we build 2 triangles on each iteration. On a little note, since we will store this triangle data in an array, this array needs to store the indices of the vertices of each triangle, and because every triangle is formed by 3 vertices, we will need an array that has

$$(2MN-2M-2N-2)*3 = 6MN-6M-6N-6$$

positions.

Once we have built the triangles, let's proceed with creating the vertices. We mentioned this before but didn't clarify much, so let's dive deeper. Let us assume that the cave is 100x100 tiles in size, such as in Figure 3.4 and that it's centered around the coordinate (0, 0, 0) of our virtual world. We will start placing our vertices on half the width to the left (negative x axis), and half the height closer to the screen (negative z axis). Then, we will start moving them further into the screen (positive z axis), and to the right (positive x axis) as we iterate and their index increases. How much we will move our vertices down the axes is controlled by a parameter called resolution, but for now, let's assume it is equal to 1 (that is, we move the vertices 1 unit per iteration). So, the x and z position of our vertices will be determined by:

$$x = (-width)/2 + column * resolution$$

$$z = (-height)/2 + row * resolution$$

Note that the *height* parameter here is referring to the height of the 2D grid - which we will call *heightmap* from now on - in 2 dimensions, which in the actual virtual world, is stored in the z component of the vertex's position, which is the depth axis. It is important not to confuse this height with the height of the vertices (y component of their position), which is fixed as a constant height of the walls.

In order to obtain the column and the row a vertex is in out of its index, we simply need to calculate [1]:

[1] Note: the // symbol is used to indicate that it is an integer division, and thus, we only take the quotient and not the remainder. The % symbol indicates a modulo operation, in which we only take the remainder of the division between the two numbers.

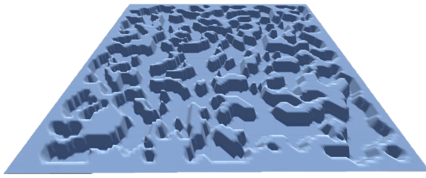

Figure 3.12: Example of a mesh build using the proposed method. Note that we built triangles directly between wall cells and floor cells, so walls are inclined, although what we could have done is having vertices at floor and wall height where cells are walls, and connect those vertices instead, thus leading to straight walls. However, this method gives already good enough results.

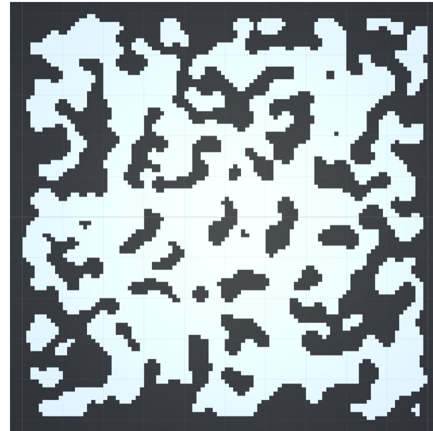

Figure 3.13: And as we can also appreciate, it matches with the heightmap we previously produced.

$$column = index \;//\; height$$

$$row = index \% height$$

Lastly, for the height of our mesh's vertices will be directly equal to a wall height parameter.

If we apply everything that we have just explained, we will obtain the result shown in Figure 3.12:

It is worth noting that, even though this mesh is 3-dimensional, the map itself is still 2-dimensional, and thus, it would need to be used in a 2-dimensional game. Fortunately for us, this method also works on 3 dimensions.

3.1.3 CA for 3-dimensional caves

Cellular automata work just as fine for generating 3-dimensional caves, or heightmaps in general, and once we know how to make it work in 2 dimensions, the transition to 3 dimensions is really smooth. We have to generate an initial map just as with the 2-dimensional method, only now we will have 3 dimensions, and then iterate over the map in order to smooth it out and create the wall clusters that are so characteristic of this method just as we did before.

Unlike before, though, every cell now has neighbours above and below itself instead of just to its left, right, front and back. Thus, we will have to modify the algorithm a bit in order to compensate for this. In particular, we will have to change the neighbour threshold parameter for which it decides if a cell must be turned into or kept as a wall. Since now we will have more neighbours in general, it will have to be much higher. In our case, we found that a good value for it is around 14 or 15 neighbours, since every cell now has 26 neighbours, and we found that in general, anything around half the neighbour count is a good parameter.

The biggest difference with the 2-dimensional approach is the method we chose to build the mesh. There is a big spike in complexity when trying to build a mesh of a higher dimension than the previous one, and several methods exist for this.

The method we chose is based on *voxels*. Voxels are the minimal cubic unit that compose a 3-dimensional object [4]. In our case, these voxels are used to represent the value of each point in our 3D grid.

When building the mesh following this approach, we will just need to render a cube centered around where our wall vertices are. This is quite simple if we follow what we explained above about building meshes. We just need to generate a voxel for each position of our heightmap, then render a voxel only if its associated position in the heightmap is a wall.

In order to render a voxel, we will need to build 6 quads, one for each face of the cube. Although it is worth noting that, since each position of the heightmap is now represented by a voxel, and to represent a voxel we will use a cube, the vertices of our mesh will not match with our heightmap's "vertices". In particular if we take M as the width of our cave, N as the height and O as the depth, our mesh will now have $(M+1) * (N+1) * (O+1)$ vertices. This is due to the fact that every position of our heightmap will now have 8 vertices that will compose the cube that is centered around it, but some of those vertices will be shared with the cubes that are centered around the different neighbours.

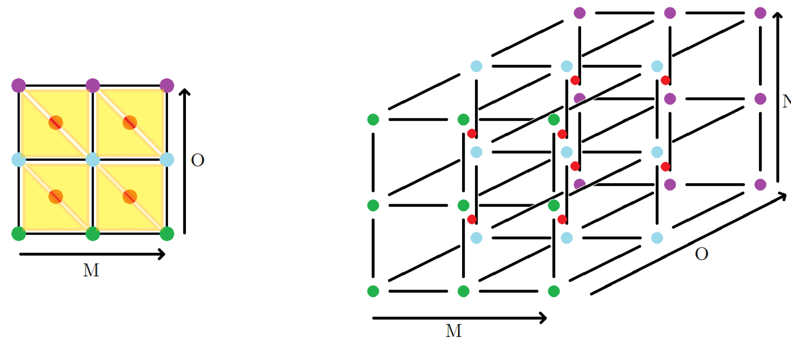

Figure 3.14: In the left hand picture, the red dots would be the heightmap positions, and the green, blue and purple dots represent the vertices of the cubes we will build for the mesh. On the right hand, we can see how this works on the third dimension as well. We can see that heightmap vertices share some cube vertices, so we don't need to create 8 cube vertices per heightmap vertex, but rather $(M+1)*(N+1)*(O+1)$, which we can check is correct, since we have a heightmap width, height and depth of 2, and a total number of mesh vertices of $3x3x3 = 27$.

This way, we can proceed to define the triangles that will form our mesh. It is easy to tell that for every position in our heightmap, we can define 12 triangles, two for each face of the cube that is conformed by them.

In our case, we order the vertices of the cubes the way shown in the picture 3.15. If we follow this convention, for example, the cube vertex with index 0 generates the triangles as follows:

- Front face: (3, 0, 1) and (3, 1, 4)
- Left face: (0, 9, 10) and (0, 10, 1)
- Back face: (9, 12, 13) and (9, 13, 10)
- Right face: (12, 3, 4) and (12, 4, 13)
- Top face: (4, 1, 10) and (4, 10, 13)
- Bottom face: (12, 9, 0) and (12, 0, 3)

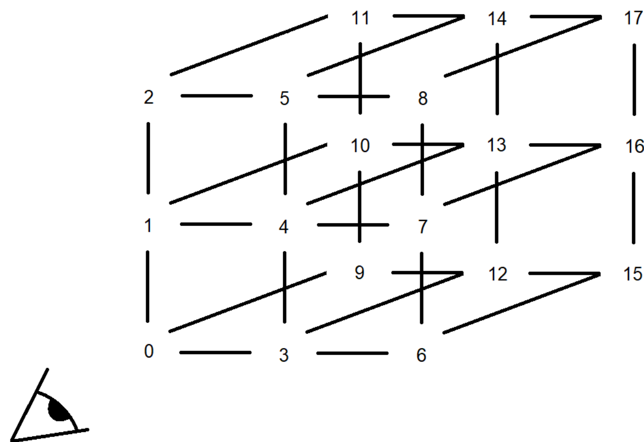

Figure 3.15: If our vertices are defined in this order, then the triangles forming the cubes that will form our mesh will be the ones commented above.

To generalize, when iterating over any vertex *i* (remember that we have the same case as in 2 dimensions, we don't build triangles on the top, right, and forward-most vertices, since they will already be part of previously built triangles), we can define the triangles as [2]:

- Front face: (i + h, i, i + 1) and (i + h, i + 1, i + h + 1)
- Left face: (i, i + w * h, i + w * h + 1) and (i, i + w * h + 1, i + 1)
- Back face: (i + w * h, i + w * h + h, i + w * h + h + 1) and (i + w * h, i + w * h + h + 1, i + w * h + 1)
- Right face: (i + w * h + h, i + h, i + h + 1) and (i + w * h + h, i + h + 1, i + w * h + h + 1)
- Top face: (i + h + 1, i + 1, i + w * h + 1) and (i + h + 1, i + w * h + 1, i + w * h + h + 1)
- Bottom face: (i + w * h + h, i + w * h, i) and (i + w * h + h, i, i + h)

If we do this for every vertex that we need to in our heightmap, we will obtain the following result:

[2]Note: the variables w and h stand for the width and height of our new vertex grid, respectively. This means that, in reality, these are equal to the width and height of our original CA grid, plus 1.

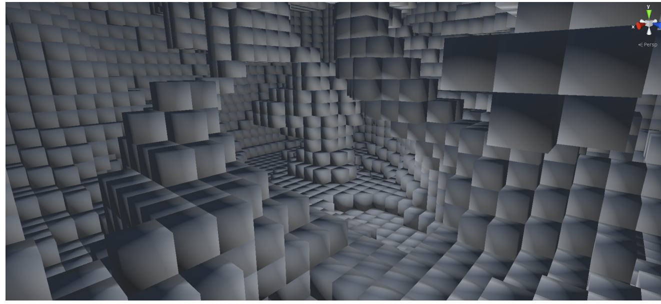

Figure 3.16: How our 3D rendered cave will look when creating a voxel-based mesh. Mind that there is a light present in the screen just so it is possible to see anything.

As we can see, the result resembles a cave in the sense that it is closed off, has multiple hallways and rooms, and has a maze-like structure.

Note that there is a little optimization trick that would save us some memory, where we only create those triangles that are seen, since, cubes that are completely surrounded by other cubes will not be seen because they are blocked by their surrounding cubes, but they will take place in memory.

3.2 Terrain generation

The problem of terrain generation is a bit different from the PCG of caves explained previously. In this part, we will introduce concepts such as noise, chunks and sewing, and we will talk about how these techniques were used in the project.

3.2.1 Noise

There are various methods to generate terrain in a procedural way [9], [11], [16]. Due to a time limitation, we have chosen the simplest one we found, although with slight modifications that yield better results. The method we chose is based on a mathematical concept called **noise functions** [17]. Remember this name, it will stick with us.

Noise functions, or more formally know as gradient noise, are a type of functions that generate a lattice of random gradients, which are then interpolated to calculate the values of the points in between the lattices. These functions are interesting for procedural generation mainly because they are pseudorandom functions, so the values seem to not follow a pattern. This is useful, as we can pick a

random point in the lattice and get a different result every (or most of the) time, thus, we can generate, in practical terms, random content.

The first known implementation of a noise function is that of Perlin noise, created by Ken Perlin [17]. The term "noise functions" was coined by him as well, and we think it is because it reminds him of the static noise from analog television sets.

As we can observe, when we set a texture with a color equal to a shade of gray the value of which is the value of the function in a certain 2-dimensional point, we get a texture that is very similar to the static noise an analog TV emits, like it's seen in Figure 3.17.

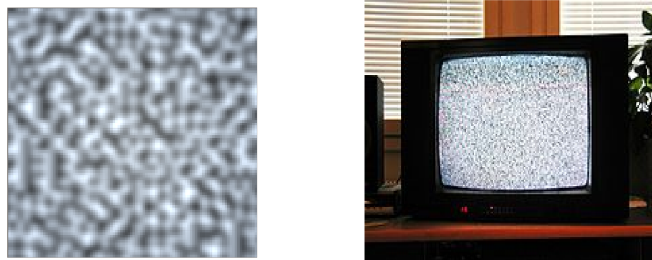

Figure 3.17: A comparison between a noise function represented as a grayscale bitmap (on the left) and TV static noise (on the right).

In this case, we used a 2-dimensional Perlin noise function provided by Unity, which uses the Mathf library [21], the implementation of which takes an x and y coordinates, and returns a value between 0 and 1 depending on the coordinates. Thus, if we set a texture where each pixel (we used a 100x100 pixel texture, 10000 pixels total) is equal to a shade of gray with its own value as its color, we get a texture that closely resembles the static noise an analog TV makes. It is worth mentioning that this implementation in particular always returns a fixed number (0.4652731) when x and/or y are integers, so we will need a parameter called scale to control this.

The heightmap's vertices were just integer numbers, so this opens a problem if we just wanted to plug the heightmap's vertices as the coordinates of our noise function, due to its aforementioned limitation. Consequently, we will need to modify the heightmap's vertices if we want to pass them as parameters to the function and get good results, and the scale parameter allows us to do precisely this. We just need to divide the vertex index by the scale in order to get an acceptable result. Note that, no matter what scale we choose, this problem won't

disappear completely, it will just get more and more disperse and thus it will be less noticeable.

Given that the operation we use to modify our original indices in order to pass them as coordinate parameters to our noise function is a division, we can deduce the effect that the scale will have in our results, and understand why it's called "scale" in the first place. Say we have a 10x10 grid. If the scale is 10, each coordinate of the function will be the index divided by 10. This means that all numbers will be closer together than the indices were originally. If we imagine a noise function like a map, this results in "picking" points out of that map that are much closer together, so the result we get is a "zoom in" effect, given that we are picking the same amount of points, but at a much smaller area. We can also see that, in this example, the only indices that will yield an integer out of that will be those that are multiples of the scale, so we can safely deduce that, in addition to the "zoom in" effect, the amount of times this integer coordinate number appears is inversely correlated to the scale, which makes it effectively disappear, especially in bigger dimensions.

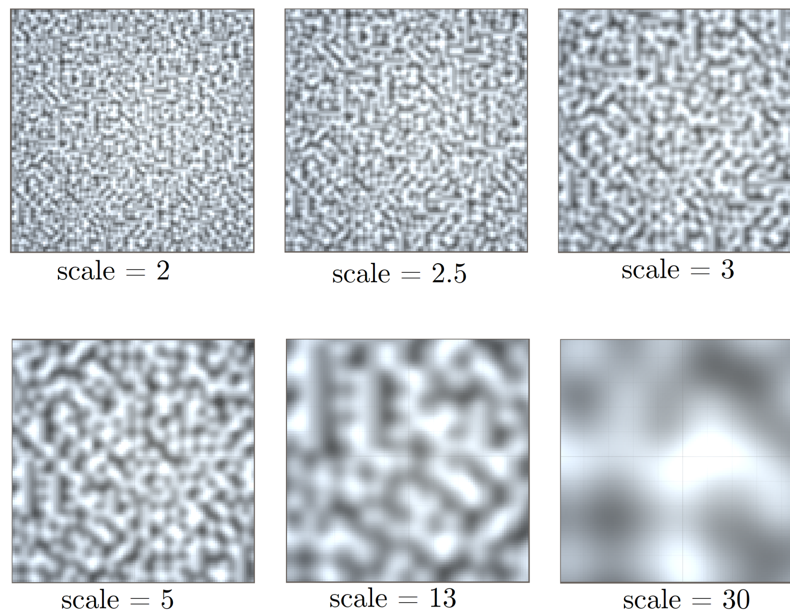

Figure 3.18: In this image, we can visually understand how scale affects the noise function. We can appreciate that it does in fact make a "zoom in" effect, in this case towards the lower left corner.

Noise functions are interesting because they represent a value for each coordinate, which we can take as a height value on a y axis. This will give us a surface relief that we can represent visually using a mesh, just like we saw for the cellular automata method.

Although a single noise function can give relatively convincing results, the power noise functions have really lies on stacking them - adding multiple together to combine their effects in different ways. We can see how this works better if we represent our function on 2 dimensions, using only 1 coordinate (x) and 1 height value (y), as we see in Figure 3.19. This can be seen as taking a cross-section of our previously seen 2-dimensional noise function and looking at it from the side.

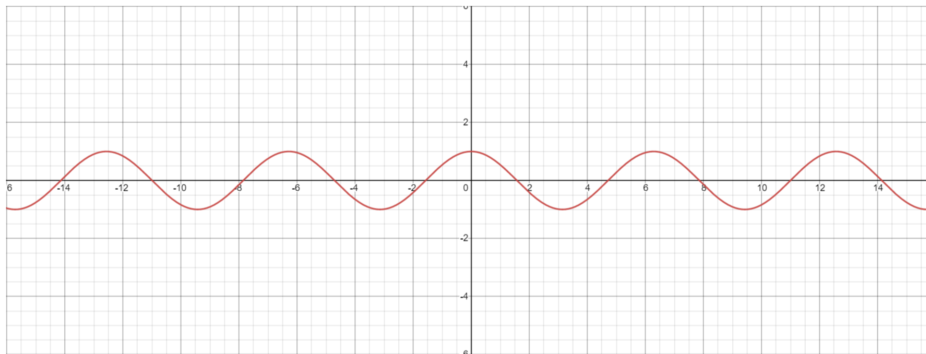

Figure 3.19: The cosine function used as a noise function. However, actual noise functions are pseudo-random functions, so they don't repeat in a pattern over the coordinates. This is just for clarification purposes.

We can observe that this function resembles the representation of a wave function, such as sound waves, or electric signals. As such, the function has a frequency and an amplitude that give it its particular shape. Apart from this, it has other characteristics such as its ability to be added to other functions, and form constructive or destructive interference, altering the final function's amplitude and frequency. For example, if we were to add the function in Figure 3.19 to a similar function with a higher frequency ($cos(5x)$, for example, seen in Figure 3.20), then we would obtain the result show in Figure 3.21.

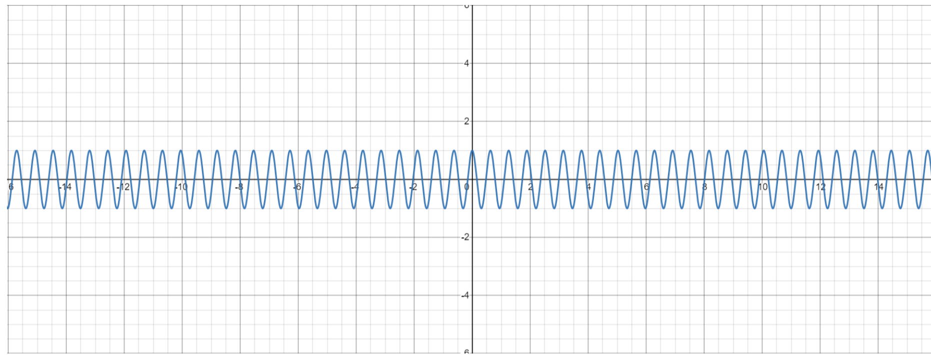

Figure 3.20: The function $cos(5x)$, used as a complementary function.

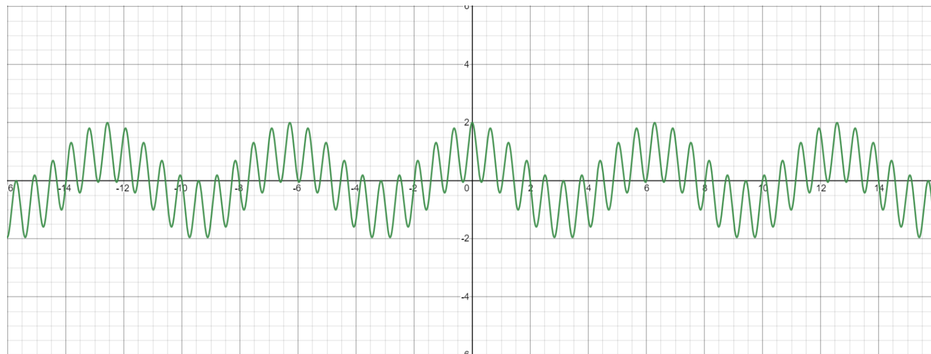

Figure 3.21: The function $cos(x) + cos(5x)$, the result of adding the functions shown in figures 3.19 and 3.20.

We can appreciate that by adding both functions together, we obtain a combination of these that has different amplitudes and frequencies than both the added functions. Furthermore, we can even add other functions with a different amplitude and phase as well to obtain different and more complex results. See Figure 3.22.

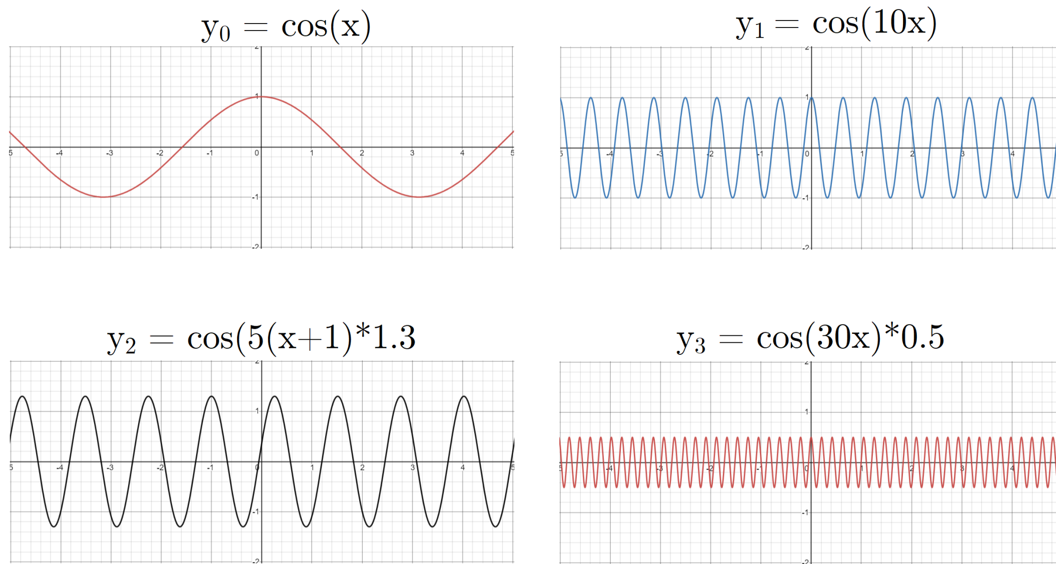

Figure 3.22: Different functions that can be used as noise functions. When using the cosine function, adding a number to the x value represents a shift in phase, while multiplying the function by a value represents the amplitude of the final wave. We can appreciate this in the figure, knowing that all four figures are in the same scale.

In Figure 3.23 we can see the functions obtained by adding the functions defined in Figure 3.22 cumulatively.

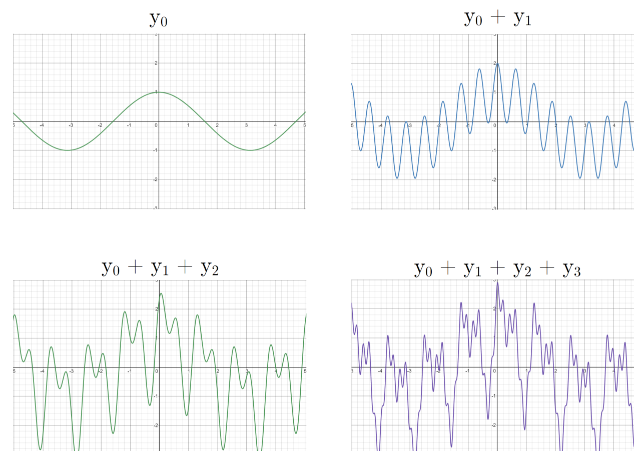

Figure 3.23: The cumulative result of adding the functions defined in Figure 3.22.

It is noticeable that the higher the amplitude of a wave, the more it will define the derived function's "general shape", whereas those waves with an amplitude on the lower side defines the derived function's "details". This is ideal for our problem, since in reality, terrain isn't always soft and smooth, and it has imperfections and details, it doesn't just conform to a "general shape" only. We then use this concept of adding different functions in order to generate a terrain that is somewhat realistic. This process is very similar to the one that was just defined, except for a few differences. First of all, for this example, we used the cosine function as our "noise function", whereas just by using an actual noise function, such as Perlin noise, we already can obtain good results that don't just follow a pattern. The other difference is how these frequency, amplitude and phase changes are calculated. In this example, they were chosen arbitrarily, but in reality, we will follow a set of rules in order to calculate them.

For this purpose, we need to introduce three new concepts, those of **octaves**, **persistance** and **lacunarity**. The first of them is very simple, the **octaves** parameters refers to how many waves we will stack on top of each other. The second and third ones are a bit more complicated and we will proceed to explain them now, but the baseline is that **persistance** controls how much the amplitude of further octaves changes, and **lacunarity** controls how the frequency of further octaves varies.

In particular, let's imagine we want to stack 3 octaves in top of each other. We will use the cosine function as our noise function for this example, but any function will technically do. We will see a different example with the Perlin noise function later on, but let's stick with the cosine for simplicity for now.

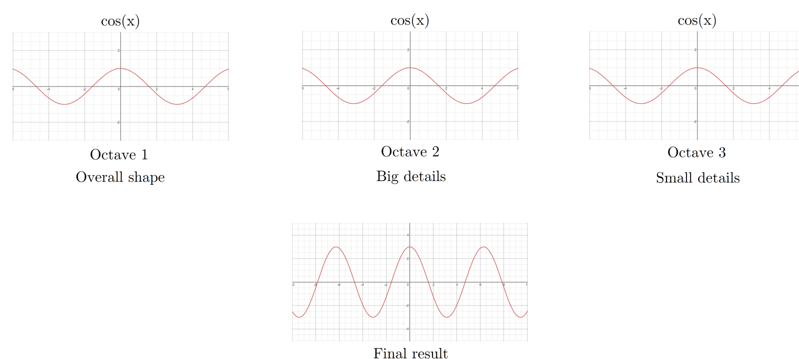

Figure 3.24: The 3 octaves we will stack on top of each other, along with a small definition, and the sum of them.

From Figure 3.24 we can see that the higher the octave, the less detail it will contribute to the overall shape of the terrain, which is the sum of all octaves. The names defined in this picture, "overall shape", "big details" and "small details" are arbitrary, and they're just a way of saying they contribute less than the previous octave. In reality, any number of octaves can be generated.

The way the lacunarity and persistance parameters will affect each octave is the following. We will start with lacunarity. Lacunarity, as we explained before, affects the "frequency" of our octaves. It achieves this by multiplying it by itself raised to the power of the octave number. Indeed, if we start counting the octaves by 0, we will need to raise the lacunarity by (n - 1), n being the octave number. This way, the frequency of the first octave (octave 1) will not be affected, since the frequency will be multiplied by 1, as lacunarity will be raised to the power of 0. In general, the octaves will follow the formula:

$$frequency_n = frequency_{n-1} * lacunarity^n \tag{1}$$

Let's apply this to our example. We will take a lacunarity of 2. See Figure 3.25.

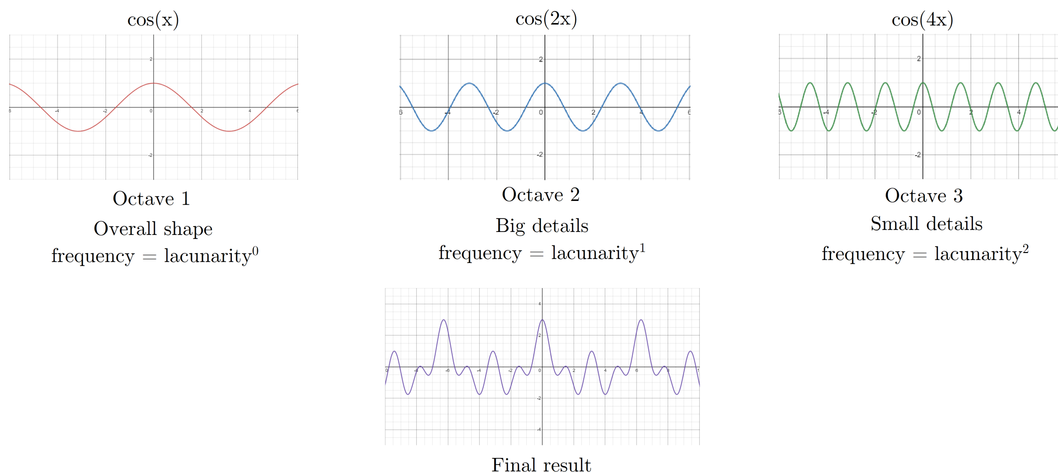

Figure 3.25: If we define a lacunarity of 2, the frequency of higher octaves increases by twice the previous octave's. For the cosine function, this is equal to multiplying x by 2 each time.

If instead of 2, we define a lacunarity value of 3, we get the results shown in Figure 3.26.

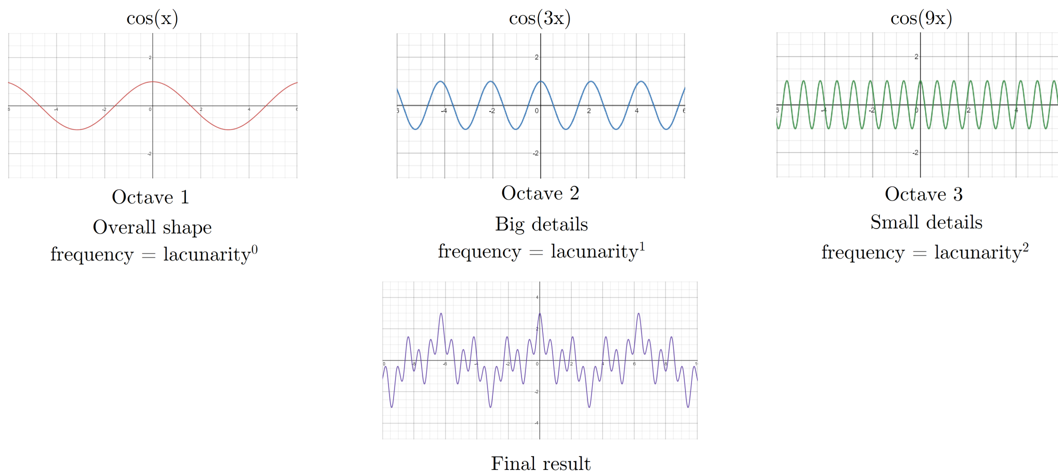

Figure 3.26: If we define a lacunarity of 3, the frequency of higher octaves increases by three times the previous octave's. For the cosine function, this is equal to multiplying x by 3 each time.

Regarding Figure 3.25 we can take that, the higher the lacunarity value, the faster the frequency will increase as we go up on octaves. Intuitively, this means that the higher the lacunarity, the finer the details each octave will contribute to the general shape. Although it is easy to reach the conclusion that a lacunarity value that is too high can "skip" over a layer of detail, and make our shape go from a smooth general surface to having very fine details, without going through intermediate details, which can lead to weird and unrealistic results. Nevertheless, this could still be an the intentional result, depending on what kind of terrain we want to recreate, but it is important to keep in mind that if this is the case, then we will need a lower amount of octaves, since the detail higher octaves will contribute will not be noticeable.

Lacunarity is a good way of controlling the exact shape we want, but we get further control over it if we add persistance. Persistance controls how much each octave contributes, in terms of amplitude. It works the exact same way as lacunarity, but instead of modifying the frequency of the function, it modifies its amplitude. Thus, the amplitude of each octave will be determined by:

$$amplitude_n = amplitude_{n-1} * persistance^n \quad (2)$$

If we apply this to our previous example, we obtain the results seen in Figure 3.27.

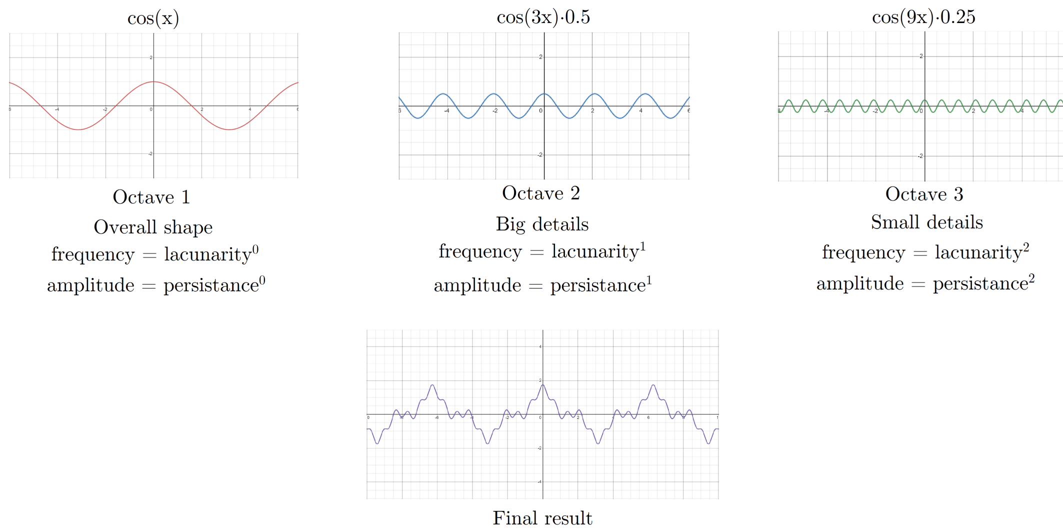

Figure 3.27: If we define a persistance of 0.5, the amplitude of higher octaves decreases by half the previous octave's. For the cosine function, this is equal to multiplying the cosine of x by 0.5 each time.

Figure 3.28 shows the same result when the persistance is set to a lower value like 0.4.

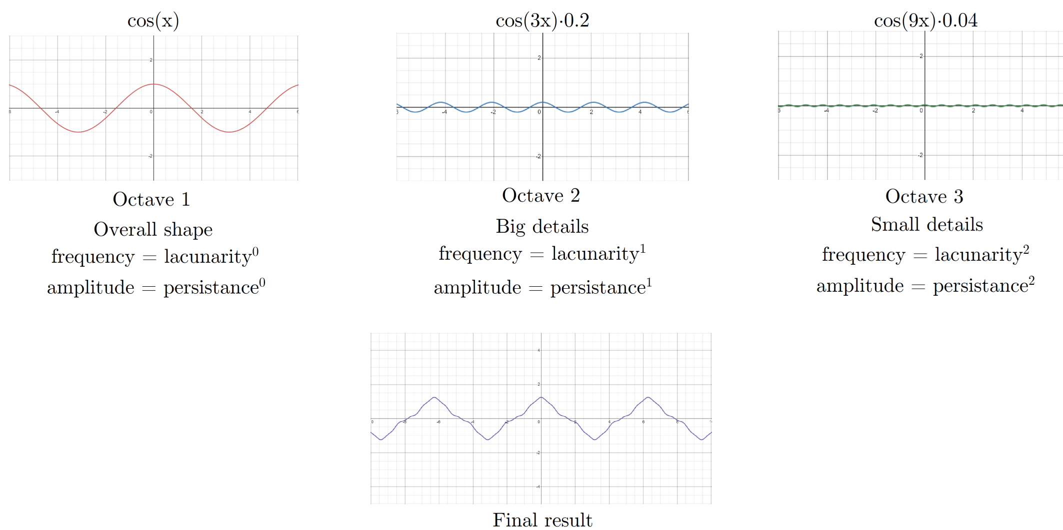

Figure 3.28: If we define a persistance of 0.4, the amplitude of higher octaves decreases by 0.4 times the previous octave's. For the cosine function, this is equal to multiplying the cosine of x by 0.4 each time.

Now that we have understood how stacking functions this way, we will proceed with how to implement these as a 3D mesh for our world. Keep in mind that even though the cosine function was used throughout the examples, we will use Perlin noise from now on.

3.2.2 Surface generation

Once we defined how noise stacking works, we can start producing our terrain. For this purpose, we will use Perlin noise. This way, we will obtain results similar to the examples we have just seen in the previous section, although they will be more distinct and will not repeat over time. We can then combine what we learned about building meshes from section 3.1.2 and construct a mesh that will resemble a 3-dimensional terrain. In order to do this, we will use a structure called *chunk* [23].

A chunk is, as it name indicates, a piece of something. It is usually used in computer jargon to refer to any piece of data that is part of a whole, for example, a section of an image or a movie. Chunks are useful since they allow for data to be split and downloaded or processed separately. In this case, we will split our terrain into chunks of a given size, and generate them separately. This gives us more control over how each chunk is being generated, and what each of them looks like.

To understand why we need chunks exactly, we need to understand how the terrain will be generated, and the problems that surface when trying to implement it via a simple approach. This simple approach would be, for example, generating a whole map stacking several noise functions directly. This would give us a terrain that is somewhat uniform, meaning that at a big scale, different terrain characteristics such as mountains, hills or plains would be distributed uniformly over the land, which might not be a desired outcome. Since our noise function will take different parameters such as scale, lacunarity and persistance, all our map would have the same parameters, and as such, all the patterns such as the mountains would be similarly generated to, let's say, the land. A modification we could think of in order to minimize this effect would be to have random values for these parameters based on the coordinates, or better yet, have a layer of noise each of these parameters take their value from, but these options don't give us much control over what is being generated precisely. This is one of the reasons we use chunks, in order to give every chunk different values for the noise function parameters, and this way have more control over what the world looks like. This being said, note that there is no "right" or "wrong" way of doing this, it all depends on what

we want our terrain to look like. It might be in our interest to give up some of this control in favor of more diverse looking scenarios, and we might actually want to implement it that way. We will see an example of this when we get to section 3.3.

Before seeing an example of how having different parameters works, we will discuss how the mesh itself is generated in this case. We will assume that our vertices are defined in the same order than we did in the case of the caves, from the bottom left corner, up and then to the right, see Figure 3.6. To achieve building the mesh, we will give each chunk a coordinate, which will help determine the absolute position of its vertices in the world. This way, the absolute position of a chunk's vertices will be a combination of their local position and the chunk's coordinates in the world. We will only be generating a surface, so the chunks' coordinates will have 2 dimensions, and the way we will distribute them will be by having our origin chunk (positioned at the center of the world) be at the coordinate (0, 0). It is not possible to distribute the chunks in the world the same way the vertices are distributed in a chunk's mesh, because, unlike a mesh, the world is virtually infinite, so there is nowhere we can start placing our chunks if we were to do it that way. So, our chunk distribution map will look like a matrix (see Figure 3.29).

(-2, 2)	(-1, 2)	(0, 2)	(1, 2)	(2, 2)
(-2, 1)	(-1, 1)	(0, 1)	(1, 1)	(2, 1)
(-2, 0)	(-1, 0)	(0, 0)	(1, 0)	(2, 0)
(-2, -1)	(-1, -1)	(0, -1)	(1, -1)	(2, -1)
(-2, -2)	(-1, -2)	(0, -2)	(1, -2)	(2, -2)

z

x

Figure 3.29: The chunk map distribution with the coordinates for each chunk. The x and z axes refer to the width and depth dimensions in the world, respectively.

Thus, we have a way of defining the position of each vertex in each chunk. The local position of each vertex depends on its coordinates (a.k.a., its row and column). We already learned how to calculate this on a previous section, see 3.1.2.

We have also learned how the chunks coordinates are made, and thus, the position of each vertex will be defined as:

$$(x,z) = (-(cs/2-1) + cc_x * (cs-1) + c, -(cs/2-1) + cc_z * (cs-1) + r) \quad (3)$$

Where cs stands for chunk size, cc_x and cc_z stand for the x and z components of the chunk's coordinate, respectively, and c and r stand for column and row, respectively.

Let's break this formula down. We will start with the $-(cs/2-1)$ part. Because our chunks are square and have an even size, they are not centered in the world. If we have a chunk size of 32, for instance, chunk (0, 0) will go from -15 to 16, since there is also coordinate 0. This is why we subtract $(cs/2-1)$ in order to begin at the left-most part of a chunk, instead of just $cs/2$, although this is arbitrary and could be changed.

Next, the part where we add $cc_x * (cs-1)$. This part here is adding the chunk size times the x coordinate to the initial vertex coordinate. What this does is, it moves the position of the vertex to fit the coordinates of the chunk it's in. This part and the previous one set the absolute coordinate. For example, vertex 0 of chunk (0, 0) will be positioned at $(-(32/2-1) + 0 * (cs-1) + 0, -(32/2-1) + 0 * (cs-1) + 0) = (-15, -15)$, and vertex 0 of chunk (1, 0) will be positioned at $(-(32/2-1) + 1 * (cs-1) + 0, -(32/2-1) + 0 * (cs-1) + 0) = (16, -15)$. This is correct, since the chunk (1, 0) needs to start at the same coordinate the chunk (0, 0) ends, for the terrain to be sewn together and not have gaps.

The last part of the formula, where we add the column and the row to the x and z axis respectively, is the part that moves a vertex according to its local position. Remember that the row and column depend on each vertex's index, and are thus local to each vertex, while the chunk size and chunk coordinate are shared between all vertices of a chunk.

Lastly, since the vertices are defined in the same order as they were when the cave mesh generation was explained, the triangles of our terrain's mesh will be the exact same.

If we apply all this to generate our terrain, we will obtain the result shown in Figure 3.30.

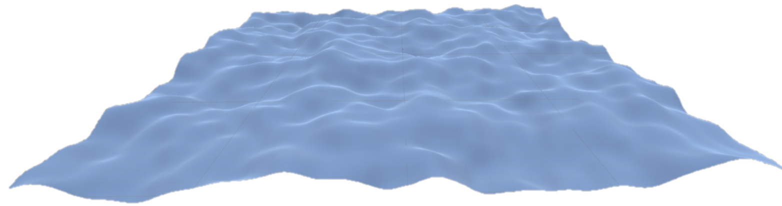

Figure 3.30: We can observe how our mesh has been created.

We can paint each chunk with a random color to appreciate their limits better in Figure 3.31.

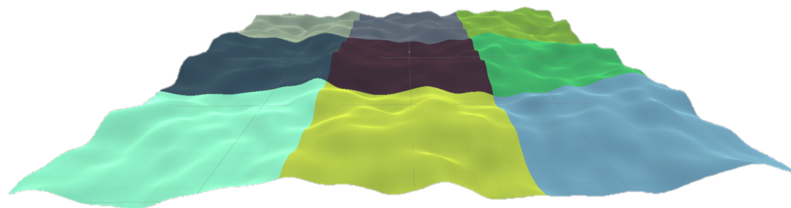

Figure 3.31: By assigning a random color to the mesh generated by each chunk, we can better appreciate what is being created by each of them.

Chunk sewing: first version

In Figure 3.31 we can distinguish how the terrain is generated in a somewhat uniform way when giving all the chunks the same noise parameters. One solution to this is increasing the noise scale, so that all the valleys and peaks are more spread apart, and increasing the vertical scale, which is a parameter that controls how the height value is calculated, basically multiplying the original height value times the vertical scale, effectively making valleys appear lower and peaks appear higher. As we can see in Figure 3.32, this is one of the ways to make the terrain more varied, but it has a drawback; some quality could be lost. Since the scale is bigger, there is less resolution from the original noise map and thus less detail on the terrain mesh, even though the number of vertices in the mesh stays the same. This is the same as zooming in on an image. If we look at an image on a 1920x1080 resolution screen, and then zoom in, we will lose some detail from the original image, as now we're only sampling a portion of the original image's resolution, even though the number of pixels on the screen hasn't changed. Another problem this methodology has is that all the terrain features will look similar to each other throughout the terrain. This means that all valleys will look similar to each other, all mountains will look similar to other mountains, and so on.

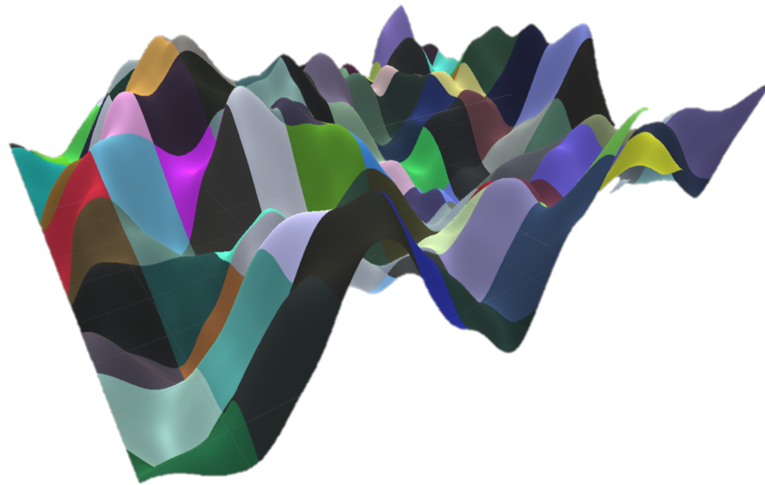

Figure 3.32: Generation with a much higher noise scale and vertical noise scale. It is clear how all the terrain looks very similar to each other. Also, because of what was just explained, in order to see the effects in place better, the number of chunks generated was increased from 9 to 141, and the chunks are bigger as well, 128x128 instead of 32x32.

To solve this problem, we give each chunk a different set of parameters so that they can sample different values from the noise function as the chunks around them.

Figure 3.33 shows the results obtained following this idea and also giving random parameters within a range to each chunk.

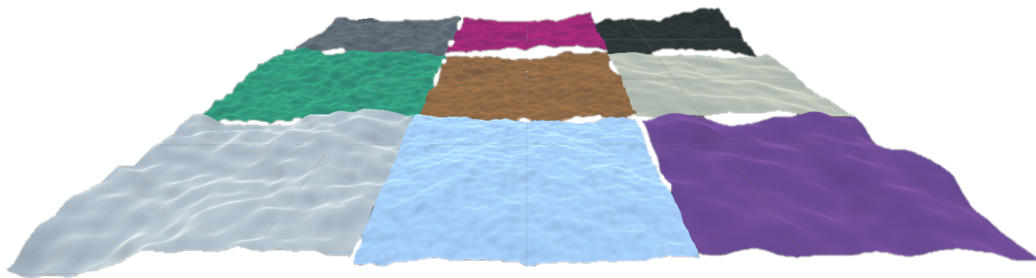

Figure 3.33: The terrain generated by giving each chunk random parameters (within a range).

Although this works somewhat better, especially when finding a good range for the randomly generated noise parameters, there is a clear problem with it.

As a consequence of having different noise parameters for each chunk, their vertices don't line up with each other as they did when they all shared the same parameters. To make a comparison, it would be like taking an image, then dividing it by pieces of the same size, then modifying each piece of the image randomly by either zooming in, zooming out, or changing the colors by processing methods such as increasing the contrast or the saturation. Unlike an image, though, our noise map doesn't hold any meaning in itself, and thus, also unlike an image, we can just sew the chunks together and get a satisfying result, since the only aspect that matters is that there is a smooth transition between chunk and chunk (in the comparison, between piece and piece), and the actual values aren't as important as they are within an image. See figure 3.34.

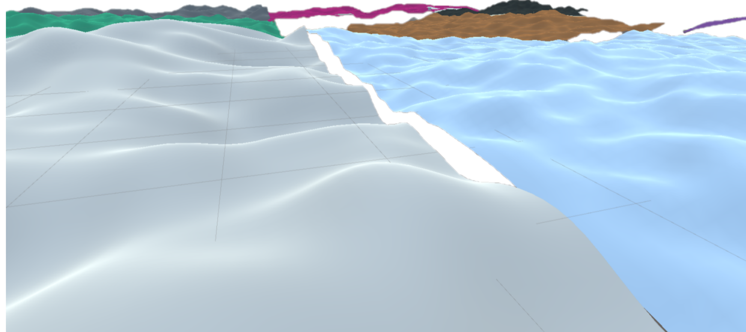

Figure 3.34: Clear gaps between chunks generated with random noise parameters.

For this purpose we will use what we have called *chunk sewing*. Sewing is the act of "joining, fastening, or repairing (something) by making stitches with a needle and thread or a sewing machine". We can deduce then why we decided to call this process sewing the chunks, since we are going to be attaching them together in order to "repair" the junctions between them.

The process itself is relatively simple, although it was quite a puzzle to come up with it, or rather, coming up with how to make everything fit together, but once explained it is easily understood. For it, we created different types of chunks:

- Terrain chunks. These are the chunks that hold the terrain information itself.
- HS chunks. Stands for Horizontal Sewing chunks. These chunks are the ones in charge of sewing the gaps between terrain chunks on the x axis.
- VS chunks. Stands for Vertical Sewing chunks. These are in charge of sewing the gaps between chunks on the z axis.

- DS chunks. Stands for Diagonal Sewing chunks. These will sew the gaps that form when HS and VS chunks are made (crossing areas).

We will explain a first version that was used when designing sewing chunks, and then explain an upgrade that makes the process much better in terms of quality.

Before building them, we must define a new parameter, which is the **sewing size**. The sewing size refers to how big the sewing chunks will be in size. In the case of the HS chunks, this means they will be a 2-dimensional array of dimensions CxS_x, where C is the chunk size, and S_x is the sewing size on the x axis. Note that, even though we are able to define different sewing sizes for the x and z axis, these must be constant throughout all of that dimension, or at least equal to all the other rows or columns, respectively. For example, the S_x can be 10 and the S_z can be 20, but they must stay like that throughout the z and x dimensions respectively, such that all HS chunks of a given row have the same size, and all the VS chunks on a given column must have the same size. It is theoretically possible for this not to be true, but the calculations become extremely complicated, and thus we have decided not to implement this for the project.

Knowing this, it becomes quite obvious that by using sewing chunks we need to move the terrain chunks over in order to make space for the sewing chunks. Thus, we need to recalculate the position of every vertex based on their local position within the chunk, the chunk's coordinate, and now also the sewing size. Luckily, we can keep the chunk coordinate map definition we used previously, even while using sewing chunks, and still be able to calculate their vertices' positions. We can then create a new coordinate map for our terrain chunks, and our HS, VS and DS chunks. See Figure 3.35.

(-1, 1)	(-1, 1)	(0, 1)	(0, 1)	(1, 1)
(-1, 0)	(-1, 0)	(0, 0)	(0, 0)	(1, 0)
(-1, 0)	(-1, 0)	(0, 0)	(0, 0)	(1, 0)
(-1, -1)	(-1, -1)	(0, -1)	(0, -1)	(1, -1)
(-1, -1)	(-1, -1)	(0, -1)	(0, -1)	(1, -1)

Figure 3.35: The new coordinate map with sewing chunks in them. Notice how each type of chunk has its own coordinate system (although with some consensus, such as, any sewing chunk is to the right or above the terrain chunk with the same coordinate).

If we define our coordinate system like in Figure 3.35, then our terrain chunk vertices will now follow the following formula:

$$\begin{aligned} x &= -(cs/2-1) + cc_x * (ss_x + cs - 2) + c \\ z &= -(cs/2-1) + cc_z * (ss_z + cs - 2) + r \end{aligned} \tag{4}$$

Remember the abbreviations: cs stands for chunk size, cc_x and cc_z stand for the x and z components of the chunk coordinate respectively, ss_x and ss_z stand for the sewing size on the x axis and the sewing size on the z axis, respectively, and c and r stand for column and row, respectively.

When building the HS chunks, for each pair of chunks in the x axis, we will take the right-most column of the left chunk, and the left-most column of the right chunk. Then, we will connect them directly using a straight line. Note that the architecture we have chosen does not allow us to access a chunk's info from a different chunk, so the map generator is the one in charge of giving the sewing chunks this information. One might think that the sewing chunks don't need to get the vertices from the left and right chunks, since it is enough to have their co-ordinates, and then they can calculate the vertices' positions. Although this is true, the sewing chunks would still need the noise parameters of both chunks in order

to calculate the vertices' positions, and this might be computationally expensive depending on the number of octaves the chunks have, so we have decided to just let them store it in memory.

More precisely, the vertices of our HS chunks will be defined as:

$$
\begin{aligned}
x &= cc_x * (cs - 1) + cs/2 + c + cc_x * (ss_x - 1) \\
y &= y_l * (1 - c/(ss_x - 1)) + (y_r - yl) * c/(ss_x - 1) \\
z &= cc_z * (cs - 1) - cs/2 + 1 + r + cc_z * (ss_z - 1)
\end{aligned}
\tag{5}
$$

where y_l is the height of the original left vertex on that vertex's row and y_r is the height of the original right vertex on that vertex's row. This ensures that the height will be correctly spaced according to the sewing size in the x axis, since it is interpolated between the values of the left and right chunk. We will get get more into this interpolation when we get to the explanation of the second iteration of this method. See Figure 3.36 for a clearer visual.

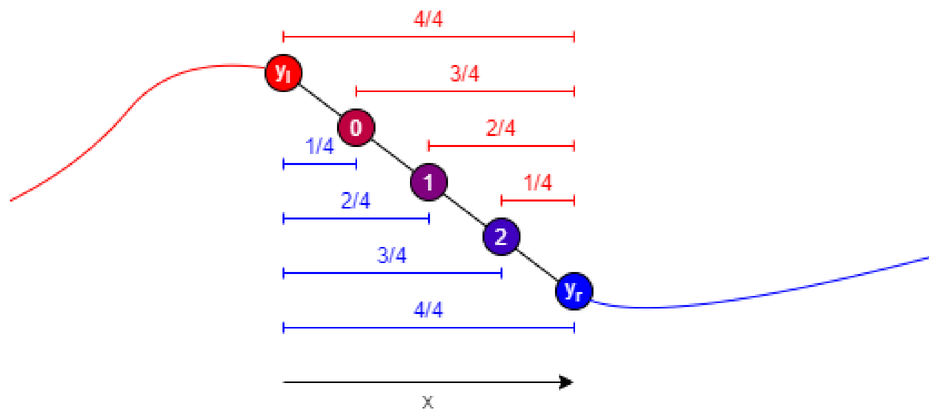

Figure 3.36: Example using a sewing size of 5, where we have 3 new vertices and the 2 vertices from the terrain chunks (in red for the left chunk and blue for the right one). We can also see the values of $(1 - c/(ss_x - 1))$ and $c/(ss_x - 1)$ above the vertices, in red and blue, respectively. If we look at the formula 5, we deduce that, the closer we are to the left chunk, the higher the number that height will be multiplied by, and thus, the closer it will be to that value. As we progress to the right, the left height multiplier gets smaller, and the right height multiplier gets bigger. The colours of the sewing vertices also help illustrate this.

This way, the vertices are positioned at the bottom left part of their chunk, plus an offset in the x direction that is equal to the sewing size on the x axis times

the chunk coordinate. This is due to the fact that, for chunk (1, 0), for example, its starting x coordinate will be half a terrain chunk plus a whole terrain chunk, plus a sewing chunk, towards the positive x. This is arbitrary and it is this way because of how we chose the coordinate system for the sewing chunks, it would be different if we had a different coordinate system. The *segment* part is also arbitrary, since we defined our sewing size to symbolize the number of vertices in between the terrain chunks. For example, a sewing size of 3 implies there will be 3 vertices including the vertices from the left and right chunk. The segment variable makes sure the sewing chunk's vertices are distributed with the right distance in between them.

To see if this checks out, we will calculate an example. Let's say we have a chunk size of 32 and a sewing size of 3 in both axes, and we want to calculate the position of the vertex 0 of the HS chunk at position (1, 0). Then, according to the formula 5, its position will be:

$$x = 1 * 31 + 16 + 0 * 1.5 + 1 * 2 = 49$$

$$z = 0 * 31 - 15 + 0 * 1.5 + 0 * 2 = -15$$

which is correct, since the terrain chunk at (1, 0) ends at the x coordinate of 49. Let's apply this and check the results on Figure 3.37.

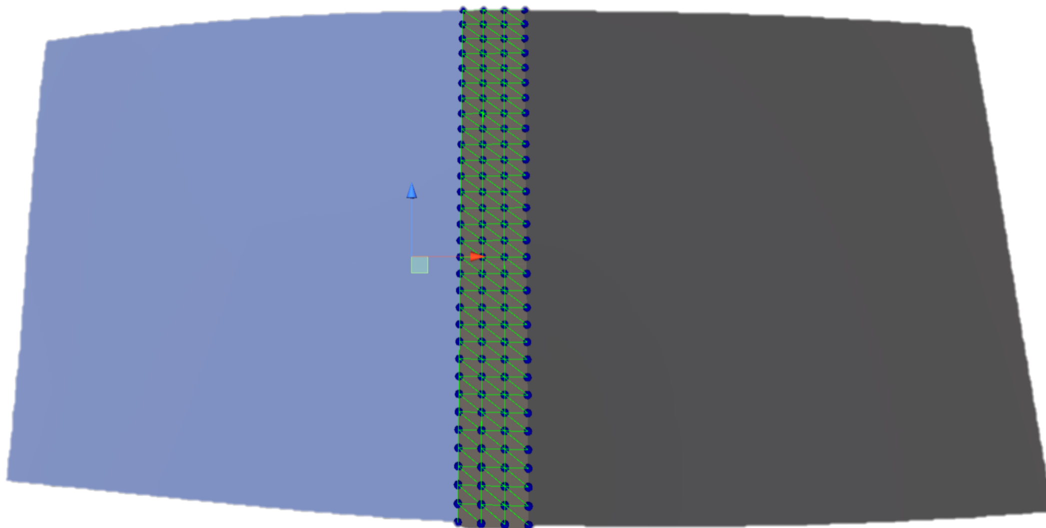

Figure 3.37: What the vertices (in blue) and edges (in green) of our mesh will look like. It is clear all their positions are correctly calculated.

We can further observe this create our mesh and give it a material. Actually, since the order of the vertex was defined the same way as everywhere else, we already know how to calculate the triangles of our mesh. We can see what this will result in on Figure 3.38.

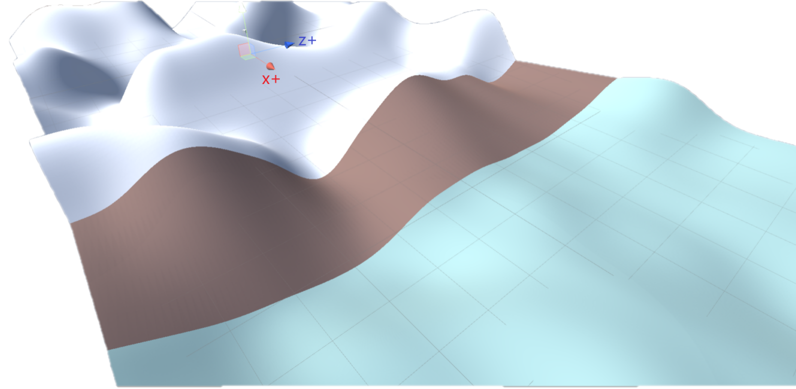

Figure 3.38: The horizontal sewing chunk (brown) between the left (white) and right (cyan) chunks. The red arrow points towards the positive x axis, while the blue arrow points towards the positive z axis. Notice how the gaps are no longer there.

In the case of the VS chunks, it is the exact same as the HS chunks, except now it's all transposed. Instead of being a CxS_x array, these will be a S_zxC array. Then, we do the exact process as with the HS chunks, but transposing everything. Thus, the positions of the vertices for the VS chunks will be:

$$\begin{aligned} x &= cc_x * (cs-1) - cs/2 + 1 + c + cc_x * (ss_x - 1) \\ y &= y_b * (1 - r/(ss_z - 1)) + (y_b - yt) * r/(ss_z - 1) \\ z &= cc_z * (cs-1) + cs/2 + r * segment + cc_z * (ss_z - 1) \end{aligned} \tag{6}$$

and y_b and y_t are the bottom and top chunk's vertex's height of the original vertex that is in the same row, respectively.

If we apply this and create our VS chunk mesh, we will obtain results such as the ones shown in Figure 3.39.

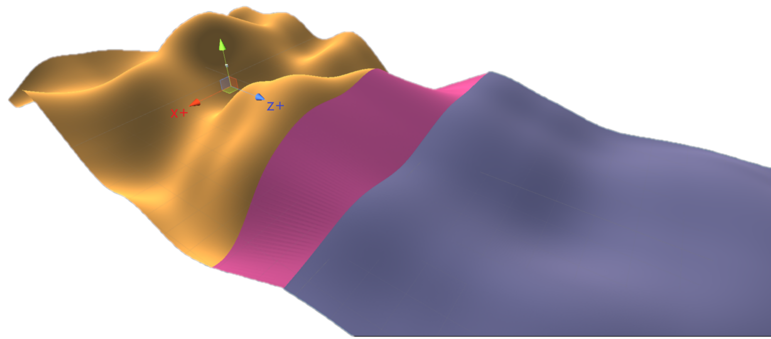

Figure 3.39: The vertical sewing chunk (pink) between the bottom (orange) and top (purple) chunks. Notice how the gaps are no longer there.

If we combine both sewing chunk types, we obtain results like the ones shown in Figure 3.40.

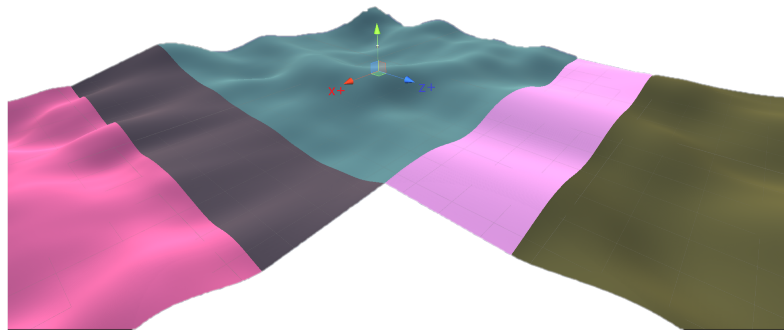

Figure 3.40: The vertical sewing chunk (light pink) between the bottom (blue) and top (brown) chunks. Also, the horizontal sewing chunk (purple), connecting the left (blue) and right (dark pink) chunks. Notice how the gaps are no longer there.

Notice how, if we were to add another set of terrain chunks, along with the HS and VS chunks between them, we'd be left with a blank space (in Figure 3.40, between the light pink and purple chunks). This is why we need a third type of sewing chunks: the DS chunks. They are sort of a combination of the HS and VS chunks. They will be a 2-dimensional array of dimensions $S_x x S_z$, as they need to fit between the HS and VS chunks. Luckily for us, the positions of their vertices will be similar to the other ones', they are defined as:

$$
\begin{aligned}
x &= cc_x * (cs - 1) + cs/2 + c * segment_x + cc_x * (ss_x - 1) \\
y &= R_1 * r/(ss_z - 1) + R_2 * (1 - (r/(ss_z - 1)) \\
z &= cc_z * (cs - 1) + cs/2 + r * segment_z + cc_z * (ss_z - 1)
\end{aligned} \tag{7}
$$

where

$$R_1 = y_{tl} * (1 - c/(ss_x - 1)) + (y_{tr} - ytl) * c/(ss_x - 1)$$

$$R_2 = y_{bl} * (1 - c/(ss_x - 1)) + (y_{br} - ybl) * c/(ss_x - 1)$$

with y_{tl}, y_{tr}, y_{bl} and y_{br} standing for the heights of the top left, top right, bottom left and bottom right chunks vertices, respectively. Notice that now, instead of a row or a column, we only have 1 vertex from each chunk, the pertaining corner vertex. Again, we will explain this process of interpolation more in detail on the second iteration, all we need to know for now is that this is called a linear (or in the case of the DS chunks, bi-linear interpolation).

Now, if we apply them all together, we can see how the DS chunk sews the gap created by the HS and VS chunk, and connects the terrain chunks diagonally. See Figure 3.41.

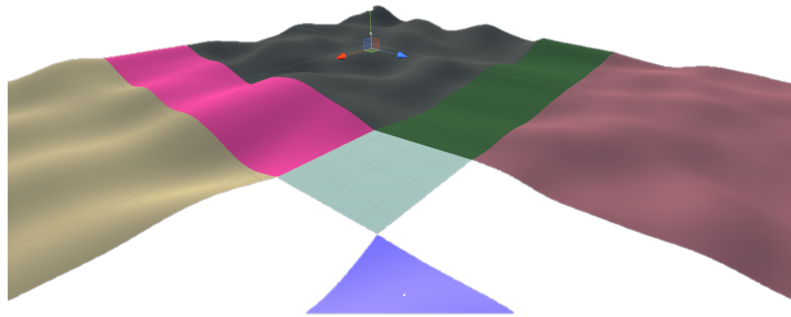

Figure 3.41: The vertical sewing chunk (green) between the bottom (black) and top (bordeaux) chunks. Also, the horizontal sewing chunk (pink), connecting the left (black) and right (beige) chunks. Lastly, the DS chunk (cyan) sews the gap left in between the HS and VS chunks, and diagonally connects the blue and black chunks.

There are no more gaps left. Although there are a couple of problems with how this sewing method works, and they most likely didn't go unnoticed.

First of all, the sewing chunks just take a straight path from each vertex of the left chunk (in the case of HS chunks) to its parallel vertex in the right chunk. This obviously works, but obviously as well, it isn't very realistic or beautiful.

One might have also noticed the second problem, which is related to how many vertices the sewing chunk's meshes have. This is a consequence of using a linear interpolation while only connecting only 1 value per row or column. Our interpolation always gives a straight line between both points, and thus, for the HS chunks for instance, we could have just built 2 columns of vertices (the left and right chunk's) and connected all the points via triangles instead of creating all those extra vertices, and the results would have been the same. Luckily for us, we were able to reuse that code when developing the second version of this sewing method, in which linear interpolation made a lot more sense.

Sewing chunks: second version

In order to get realistic sewing between chunks, instead of connecting the last column, row, or corners, we must interpolate every point of the map precisely. This means that every sewing chunk will store a portion of its adjacent chunks map, instead of just the last column, row, or corners. Then, if we interpolate the values like we have done previously, we will obtain a real interpolation of the maps.

Let's show an example. Using a chunk size of 32 and a sewing size of 3 in the x axis. Let's say we want to sew the terrain chunks (0, 0) and (1, 0), so we will be working on the HS chunk at (0, 0) (of its own coordinate system, see Figure 3.35). In the first version of this method, which was explained previously, we assumed that the map manager passed the right-most and left-most column of the left and right chunks, respectively. This time, instead of just that, it will pass an entire portion of a noise map that is an extension of each chunk in the direction that is needed. For instance, the HS chunk (0, 0) will receive a 32x3 array that contains an extension to its right. This means that if the left chunk originally ended at $x = 16$ (since the chunks are centered around their coordinate, so terrain chunk (0, 0) would start at $x = -15$ and end at $x = 16$), then the HS chunk will receive a noise map from $x = 16$ to $x = 47$, which would be a continuation of the left chunk's map if it were to extend to the right. The same is true for the right chunk. This might be confusing at first, but it's really simple to understand in reality, hopefully Figure 3.42 will clear up any confusion.

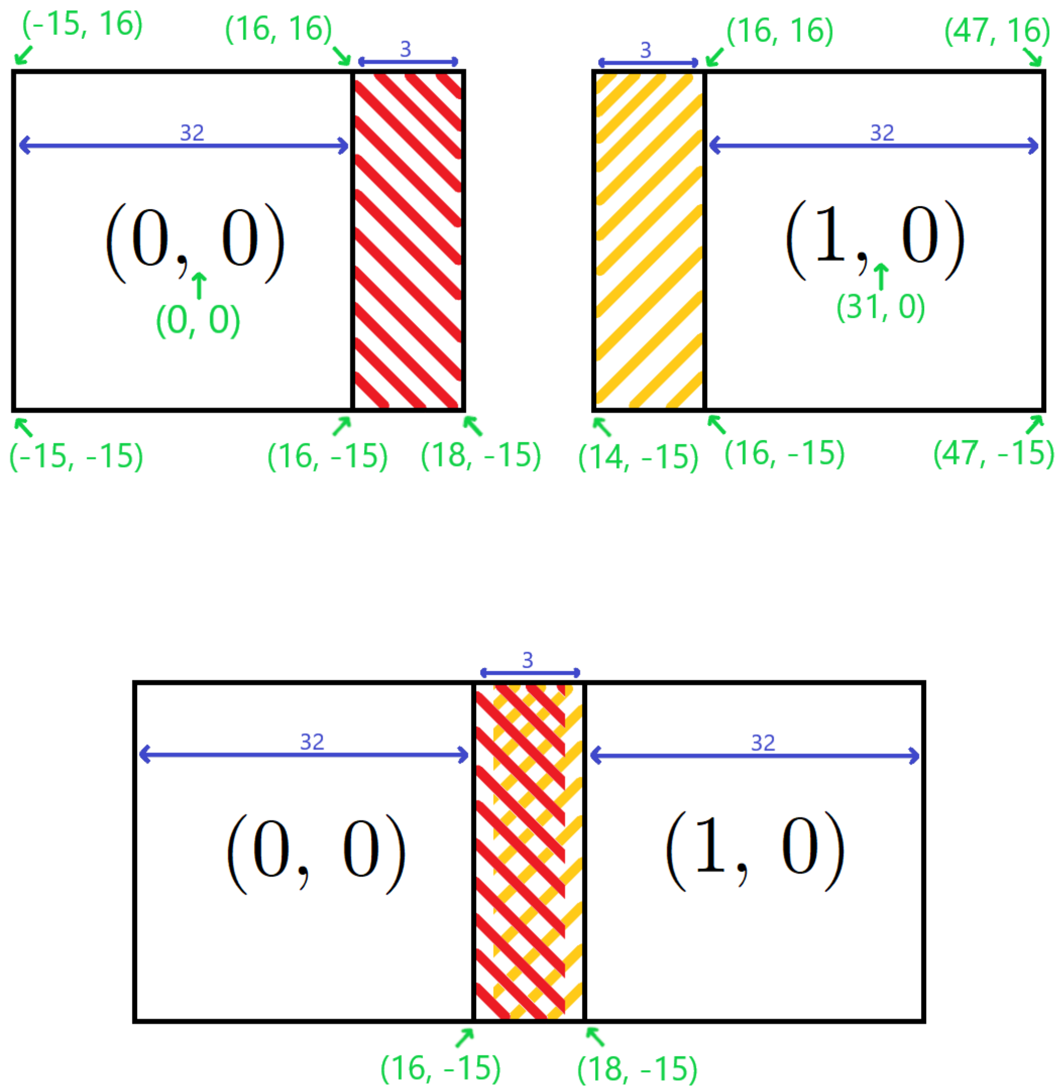

Figure 3.42: The HS chunk (0, 0) will be sewing the terrain chunks (0, 0) and (1, 0). The green arrows and coordinates represent coordinates where the chunks would end normally, if there was no sewing involved. We generate an extra portion to the right of the left chunk (in red stripes), and an extra portion to the left of the right chunk (in orange stripes), and then we stitch them together, moving the terrain chunks from their natural positions (in this case, we only move the terrain chunk (1, 0)).

Once we have this, the interpolation becomes easy, all we have to do is apply the same formula we did before, only this time we have the added bonus that the correlation between rows will also be high, since, due to the nature of the noise maps, every vertex is influenced by its neighbours. As we see in Figure 3.42, the influence of a chunk is bigger the closer it is to it, and then it blends towards the middle. Note that the only modification we need to make to our sewing method is changing the type of one of the variables in how the height is calculated, so in the formula

$$y = y_l * (1 - c/(ss_x - 1)) + (y_r - yl) * c/(ss_x - 1) \tag{8}$$

y_l and y_r will just be 32x3 noise maps, instead of 32x1 as before. So, in this case, every vertex takes the height of its own index in both maps, instead of just the vertices in its same row. For example, the sewing chunk's vertex on index 22 will take y_l as the vertex with index 22 on the left chunk's noise map, and y_r as the vertex with index 22 on the right chunk's noise map, instead of taking y_l and y_r as the vertex in the row 7 of the left and right chunk's noise maps, respectively.

Applying this sewing method to our HS chunks gives the result shown in Figure 3.43.

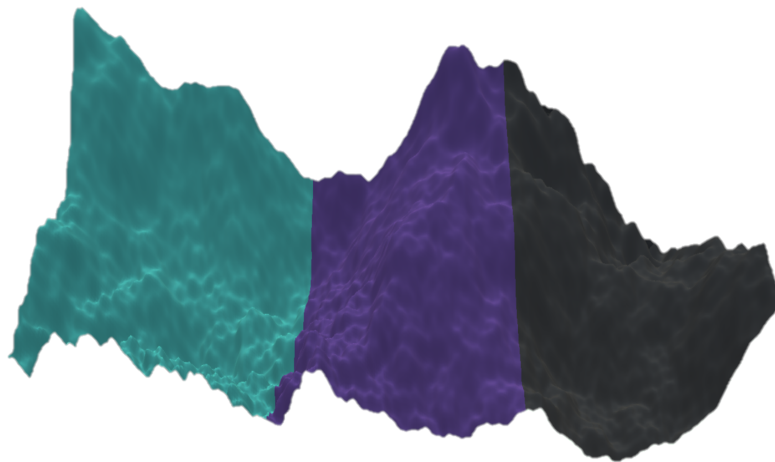

Figure 3.43: Horizontal interpolation between a left (cyan) and right (black) chunk. The difference with the previous method is very noticeable, as the latter would have just drawn straight lines between them, whereas this one blends the two together seamlessly.

For contrast, Figure 3.44 shows the results we would have obtained on similar chunks with the first version of the sewing method.

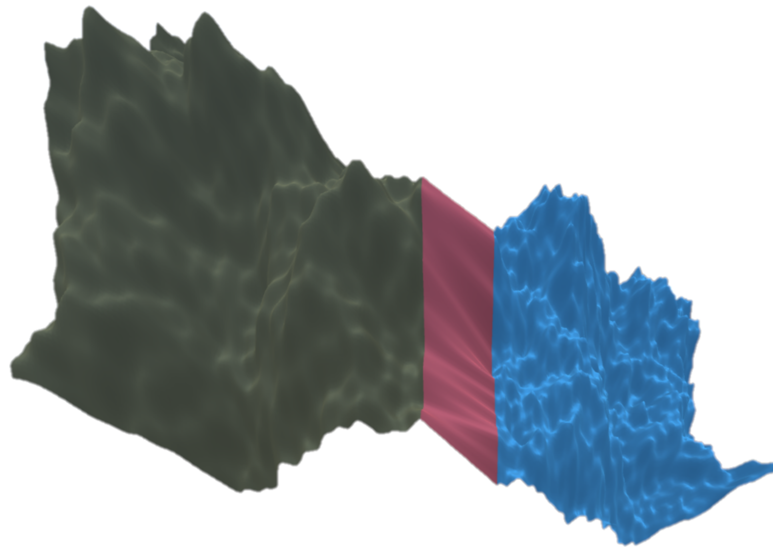

Figure 3.44: The difference between the different sewing methods is clear, especially on mountain terrain, or in general, any terrain with a reasonable amount of detail.

Furthermore, applying this method to all our types of sewing chunks gives the results seen in Figure 3.45.

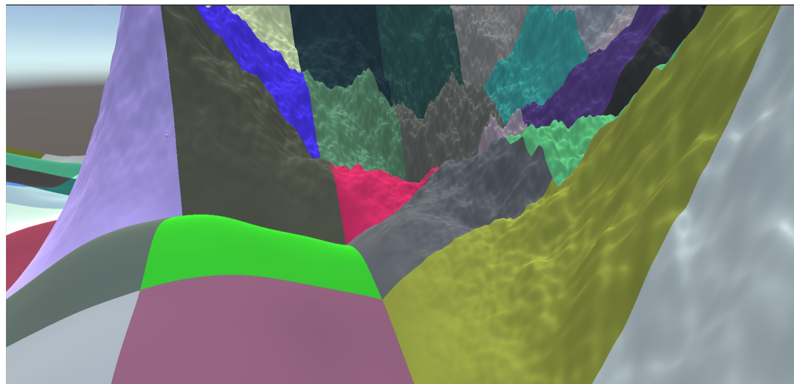

Figure 3.45: Applying the new sewing method to all our sewing chunk types. We can see how the integration between very distinct terrain chunk types is very smooth as well.

That's all when it comes to the chunk sewing section, although we will expand upon it when we talk about materials and textures!

Real time generation

In this section, we will talk about how the real time generation works. It is a very simple task but an important nonetheless. The juice of the matter will come when we talk about the optimization of this process, but this section will just be about explaining how it works.

Essentially, a map manager is in charge of creating and destroying chunks as the player moves, since, recall that the player has to be able to move around the world, and thus we need to create new chunks and destroy old ones as they move. We will spawn a number of chunks initially in a square of a certain area around the player, and create new chunks and destroy old ones when the player moves to a different chunk.

At the beginning of the execution, a number of terrain, HS, VS and DS chunks are spawned in a square around the player. The number of chunks that spawn is a tweakable parameter, but note that the only number this will affect directly is the number of terrain chunks generated, since the number of all sewing chunk types depend on this. In particular [3],

$$|HS| = (|T|_x - 1) * |T|_z$$

$$|VS| = |T|_x * (|T|_z - 1)$$

$$|DS| = (|T|_x - 1) * (|T|_z - 1)$$

This is clear if we look at Figure 3.35.

In order to update the terrain as the player moves, we keep track of several things:

- We will obviously need to track the player position. For this, we will update it every frame, and determine whether they have stepped onto a new chunk - in order to know whether we need to load the next row or column of chunks - using a general formula that takes in world coordinates and transforms it to our chunk coordinates:

$$cc_x = (wc_x + cs/2 - 1 + ss_x/2)/(cs + ss_x)$$

$$cc_z = (wc_z + cs/2 - 1 + ss_z/2)/(cs + ss_z)$$

[3] The | symbols are used to indicate cardinality, that is, the number of chunks.

Note that for this to work, the division must either be a whole division, or rounded up, since precisely, we need the quotient of the division, which will be our chunk coordinate. Also keep in mind that even though the term "general formula" was used, this only works if the chunk coordinates are defined the way they are in the project. The term was used to refer to the fact that it applies to any conversion we might want to make from world coordinates to chunk coordinates, not only to the player position.

-

First of all, we will obviously need to track the player position. For this, we will update it every frame, and determine whether they have stepped onto a new chunk - in order to know whether we need to load the next row or column of chunks - using a general formula that takes in world coordinates and transforms it to our chunk coordinates:

$$cc_x = (wc_x + cs/2 - 1 + ss_x/2)/(cs + ss_x)$$
$$cc_z = (wc_z + cs/2 - 1 + ss_z/2)/(cs + ss_z)$$

where

- cc_x and cc_z are the x and z components of the chunk coordinates, respectively.
- wc_x and wc_z are the x and z components of the world coordinates.
- cs is the chunk size.
- ss_x and ss_z are the sewing sizes along the x and z axes, respectively.

Note that for this to work, the division must either be a whole division, or rounded up, since precisely, we need the quotient of the division, which will be our chunk coordinate. The term *general formula* was used to refer to the fact that it applies to any conversion we might want to make from world coordinates to chunk coordinates, not only to the current player position.

Next, we will have to keep track of the chunks that are being generated. We need to store this in a list because we will need to iterate over it when deciding whether we will set a chunk to active or not. Reason being that in order to save computation time, we will store the different chunk's information so that later on we can just get it from memory and render it directly, without having to calculate it again. This list in particular will store the chunks' positions, and their data,

so we will actually use a dictionary data structure, where the positions are the keys, and the mesh data are the values. This list poses some problems, such as running out of memory when having visited a considerable number of chunks, and it would not be needed were the chunk generation fast enough.

The easiest way to implement this is by, first of all, setting all the chunks that are currently visible to inactive. Then, we will calculate which chunks have to be generated by calculating their position - remember that a number of chunks will be generated around the player, so this process is simple, all that's needed is offset the player's position in chunk coordinates -, and we will use these positions to search in the dictionary. If the position was found in the dictionary, then we will render its associated mesh. Otherwise, we will proceed to calculate its mesh data given its position. The figure 3.46 shows this more clearly.

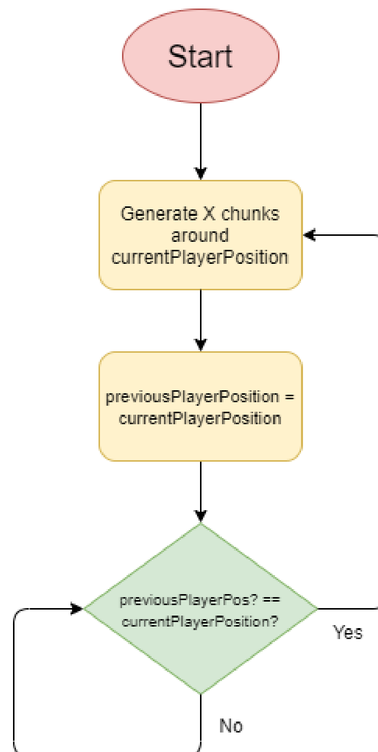

Figure 3.46: Flow chart of our algorithm. Note that currentPlayerPosition is updated on every frame, whereas previousPlayerPosition is only updated when the player changes its position. Both positions are expressed in chunk coordinates, so that the rendering of new chunks only triggers when going from a chunk to the next.

3.3 Biomes

Now we know how the map will be created and rendered as the player moves, from a programming point of view. But, how do we decide **what** is being created? What chunks are being generated where, and how exactly are these chunks being formed? Since we want a somewhat realistic world generation that adjusts as best as possible to reality, we will a biome system.

A biome is a large area that is characterized by its flora, fauna, soil and climate [2]. Examples of biomes would be a jungle, a desert, a coral reef, a tundra... and so on. Lots of video games use these biomes to create terrain with different characteristics, and thus, various techniques have been developed over the years to achieve this.

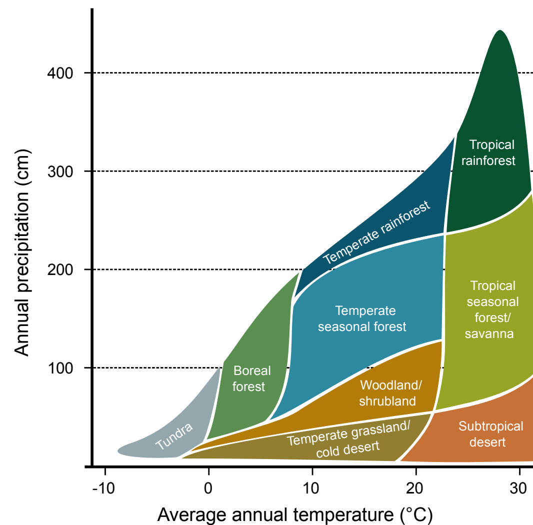

Figure 3.47: How biomes are distributed on Earth.

Our approach proposes to use a different layer of noise in order to determine which biomes would spawn where? We use another layer of noise that determines the biome that will spawn, based on arbitrary parameters. We define a range for every different biome, where, if the noise function's value in a certain coordinate fell inside a biome's range, then that biome would be spawned at those coordinates. In Figure 3.48, we show the creation of 3 biomes (mountains, hills, and plains).

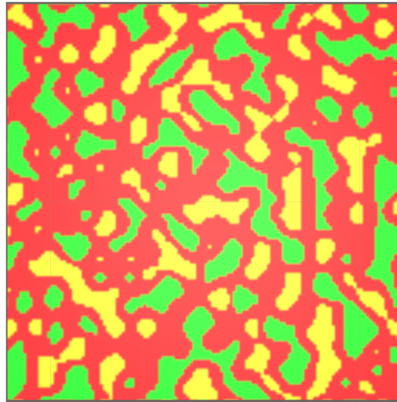

Figure 3.48: The biome map that shows how the biomes will be distributed. Note that this biome texture is 128x128 pixels, where each pixel is a chunk, so the real map that would be generated would be way bigger and every biome is thus bigger in surface. In this case, yellow indicates the presence of mountains, red indicates the presence of hills, and green, the presence of plains.

This approach is satisfying enough, although we came up with another way of making sure there was some more consistency and realism in the world. It could be extruded on the way that biomes follow certain patterns based on real physical variables that are obviously very realistic. Indeed, why not make the temperature and precipitations be defined by yet another layer of noise? This way, it won't be as realistic as on Earth, mainly because how the temperature and precipitations would be distributed wouldn't make any sense, since there is no meteorology, there are no north or south poles, and as such, they would follow a pretty "random" distribution (at least compared to Earth). Nevertheless, doing it this way ensures that there will at least be some sort of local consistency when changing biomes, since temperatures and precipitations will not change drastically over a small distance, thus, very dissimilar biomes would not appear together. This already happens with the previously presented approach, although having parameters such as temperature and precipitations makes it easier to control our world, since they're real parameters and we have multiple studies on how exactly they influence the appearance of different biomes, as opposed to having "range values" that are completely arbitrary and would require large amounts of time to experiment with in order to obtain results as realistic as the ones obtained with temperature and precipitation values. The reasons we decided to stick with the original approach are that there aren't enough biomes defined for the latter approach to be useful enough, as well as that the initial approach works well enough. One of the objectives was to get a realistic terrain, although this realism

only applies to the surface itself, and the transitions between them, but achieving an extreme realism on how the biomes are distributed actually contradicts the premise of having an endless world, since, in reality, planets are not endless, and, as we commented before, the biomes in them are distributed following physical consequences product of their topology, geology and other attributes, all of which we don't simulate.

3.4 Texturing

In order to achieve further realism, we created a set of materials that have a texture, to apply them to our surface. To do this, we created a different material for each of the biomes, each with its own texture, and a special material for each of the sewing chunks.

For the texturing to work, we need to define a custom shader for our biome materials and a different shader for each of our sewing materials. This is not strictly necessary and could all be done with the same shader, but this way the project will be more organized and easier to modify in the future.

3.4.1 Biome material shader

The shader that will be used to render the terrain chunks is very simple: we only need to take the material's texture, scale it by a parameter called scale, and our texture will be applied. Luckily, HLSL (which stands for high level shader language), the language Unity uses for custom shaders such as the ones we are making has a function that we can call in order to carry this out. So, the color of our pixels according to our shader will be calculated as seen in formula 9.

$$color = tex2D(texture, pos_{xz}/scale) \tag{9}$$

where texture is the material's texture, and pos_{xz} is a 2-dimensional vector which stands for the x and z coordinates of the point the shader is working on. As we mentioned, we use the parameter scale to change the "size" of the texture, which we do by dividing the coordinates by it.

We can see the results in Figures 3.49 and 3.50, when applying the texture seen in 3.51.

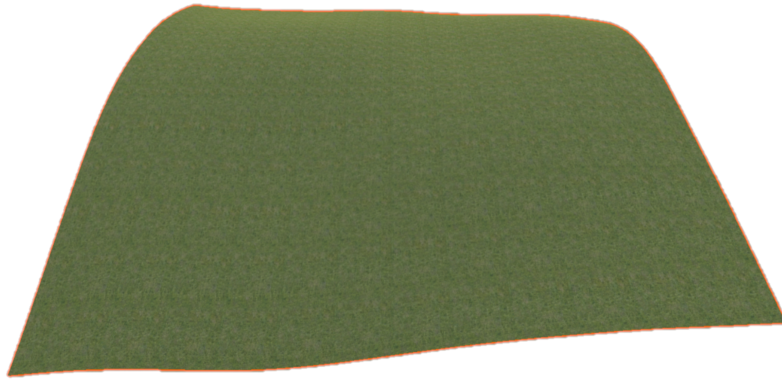

Figure 3.49: The hills material's texture applied to our terrain chunk, with a scale of 10.

Figure 3.50: The hills material's texture applied to our terrain chunk, with a scale of 42.

Figure 3.51: The texture used for the hills material.

There is a problem with this approach, though. What the *tex2D* function does is project the texture image along the y axis, since we are taking the x and z coordinates of the point. This projects the image **only** along the y axis, so the texture looks good on plain regions, but it doesn't look as good on the regions that have a big difference of distance on the projected axis' coordinates, with respect to the difference of distance on the other axes. We can see an example of this in Figure 3.52

Figure 3.52: Stretching of the texture when projecting it along the y axis like in formula 9.

As we can see, on the parts that are very vertical, there is a large increase on the y coordinate with respect to the difference in x and z coordinates, so the texture gets stretched compared to the flatter regions. For contrast, we can see the same example, but instead of projecting it along the y axis we will project it along the x axis, in Figure 3.53.

Figure 3.53: Stretching of the texture when projecting it along the x axis.

In order to solve this problem, we will implement a method called **triplanar mapping**. This method consists on finding the normalized normal vector of the

mesh at the desired point, and multiplying the projection of the texture on each axis by its component on the normal. So, the color of a vertex will be defined as:

$$color = p_x * n_x + p_y * n_y + p_z * n_z \quad (10)$$

where p_{axis} is the projection of the texture along an axis, and n_{axis} is that axis' component of the normal vector. This way, if the normal is (0.3, 0.5, 0.2), that means that the region will be mostly flat along the y axis, and so the final projection will have a higher weight associated with the projection along the y axis.

When applying this method, we get the results seen in Figure 3.54.

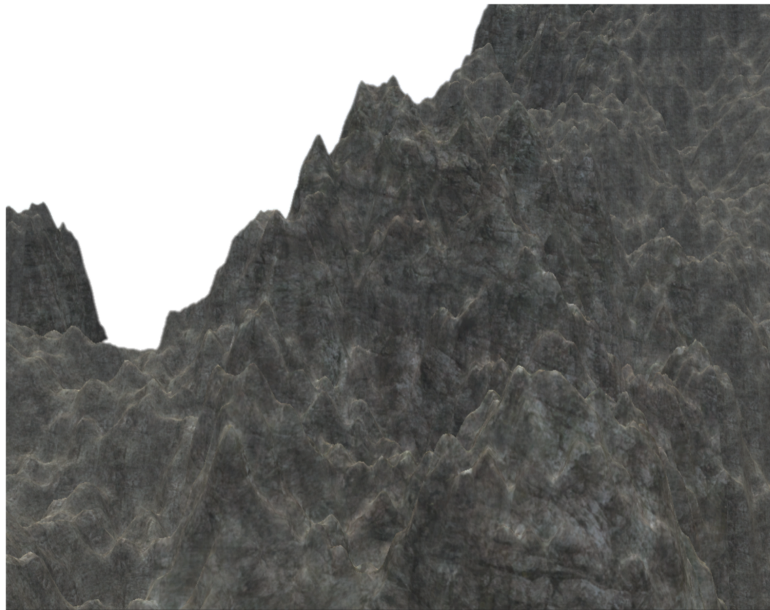

Figure 3.54: The texture mapping when applying the triplanar mapping method.

It is clear that the method works as intended, as there is no stretching along any of the axes.

3.4.2 Sewing material shaders

In the case of the sewing chunks, we can't just have a texture just like we did with the terrain chunks, since the transitions between chunks would be harsh. Instead, we need to interpolate the texture out of the different textures that compose the chunks that are being sewn. This is simple, considering we already know how to

interpolate values, since it was explained on section 3.2.2. We will do exactly the same, except now we will interpolate the colors instead of height values.

If we implement this, we get a result like the one in Figure 3.55.

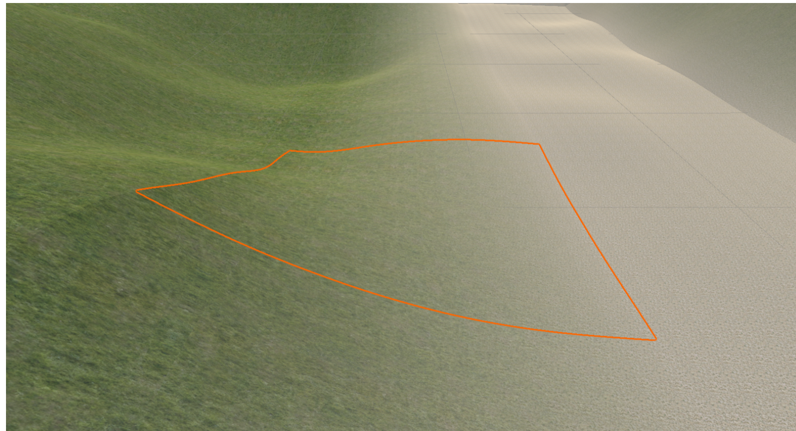

Figure 3.55: Horizontal texture interpolation between two chunks, a hills biome chunk on the left and a plains biome chunk on the right. The orange lines indicate where the sewing chunk starts and where it ends, for a better visualization.

As previously, the interpolation of the DS chunks has to be bi-linear, since they interpolate from a total of 4 chunks. Although, we can apply it just as we did before, getting results such as the one in Figure 3.56.

Figure 3.56: Diagonal texture interpolation between four chunks (the one on the top left is a mountain chunk, while the rest of them are hills). The orange lines indicate where the sewing chunk starts and where it ends, for a better visualization.

3.5 PCG adaptability

As the introduction pointed out, we wanted to make the world interactive and adaptable to the player's style. This means that somehow, we needed to make the player's actions have an effect on how the world was being generated. This didn't seem like an easy task, but if done well, it could improve the playability and the entertainment provided by the game.

The first technique we implemented was measuring how much time the player spent in each biome, to then modify which biomes were being generated. There were several problems with this, though. First of all, we didn't know how to generate the world exactly. The first idea of how to do this came to us when we were implementing the biome map with the range values. A simple approach was just to modify these range values to bring them up or down for every second

that the player spent in a biome. Although, we didn't know how to implement it exactly. It was unclear whether it would be better to make other biomes more likely to appear when the player spends a long time on a biome, or less likely. If we make other biomes more likely to appear, then we are not giving the player what they would like to visit more, although at the same time we would be forcing the player to move out of their "comfort zone". If, instead, we make them less likely to appear, then we might run into a snowball situation where the player is trying to leave a biome they've been visiting for a while, but cannot find any other biome since they're less likely to appear, and since he can't find any other, he spends even more time on the current one, which makes others even less likely to appear.

The approach we decided to take finally is to have the values crunch for the value that the player is visiting, every second. The reason is that a sort of minigame the player would be playing is trying to "chase" the biomes that they want to see, which might make world exploration more interesting.

Chapter 4

Software Design

4.1 System architecture

In this section, the system architecture will be explained. For this project, the Unity game engine was used, in particular, the 2019.2.17f1 version. Unity is based around OOP (Object-Oriented Programming), and implements an interface called MonoBehaviour. Unity uses something called GameObject, which is a class every object in the game implements. It is worth mentioning that for Unity, classes are defined on files named scripts, and every script an object implements must derive from a class called MonoBehaviour. It is possible to have classes that don't derive from this MonoBehaviour class, but they cannot be attached to a GameObject object. For example, if there is a GameObject that will be the player, which we'll call Player, and we want to attach a script to it so that it implements the class PlayerClass, then PlayerClass must derive from MonoBehaviour. Although, we only need to do this if the PlayerClass object interacts directly with the Player GameObject, such as modifying its position, scale or rotation in the world. Otherwise, we can for instance create a class called PlayerStats, which is *not* attached to the Player GameObject, and thus must not derive from MonoBehaviour. This class could hold the player's *stats* (short for statistics, the different attributes the player has), and the PlayerClass could instantiate an object of this PlayerStats class (which does not mean the GameObject Player has it attached, we will see this more clearly when we talk about Components), which could hold, for example, the speed the player will move at, and then the PlayerClass object could modify the GameObject's position in the world based on the speed defined in the PlayerStats object.

The way this works in the game is, every MonoBehaviour must implement the Start and Update methods, in case we want them to be modified at all in any way based on the events happening in the game. The first frame after the execution of the game starts, or when a GameObject is created, Unity calls the Start method for every Component, or for that GameObject, respectively. and sets their variables to the initial values defined in that method accordingly. Then, for each frame, Unity will call the Update method for every **Component**, and update their variables' values accordingly.

4.2 Components and GameObjects

A **Component** is any script that derives from MonoBehaviour and can thus be attached to a GameObject. Any GameObject can have more than one Component attached to it. For instance, our Player GameObject could have a PlayerClass Component attached to it, as well as a Renderer, which would hold and render its 3D model. Every GameObject has at least one Component attached to it, which is the Transform Component. This Transform component holds information about, and controls the position, scale and rotation of the GameObject in the world. Nonetheless, we will define and attach more Components to our GameObjects.

This project is structured in the following way, having the following Components:

- Native Unity Components:
  - **Camera:** This Component controls the camera in the 3D world. Everything that is rendered onto the screen is based on this Component.
  - **Light:** This Component is used internally to calculate the lighting on all GameObjects in the scene. This particular Light is of the directional type, and simulates a light that is very far away and points in a direction, such as the Sun.
  - **MeshFilter:** This Component holds the information about the 3D mesh of a GameObject. This mesh will then be used by the MeshRenderer.
  - **MeshRenderer:** This Component is in charge of rendering a mesh onto the screen. This Component takes the mesh from the MeshFilter Component in the same GameObject to render it.
  - **MeshCollider:** A Component which has a custom collider, shaped by the mesh of the MeshFilter Component of the same GameObject.

- Components created for this project:
  - **CameraControls:** This Component controls the position and rotation of the Camera, based on the position of the player and mouse input the player makes, respectively. It is a first or third-person camera.
  - **PlayerControls:** This Component is in charge of reading the player inputs on the control scheme and updating the player's position and rotation according to this.
  - **SurfaceGenerator:** This Component controls the different chunks that will be generated, whether they are Terrain, HS, VS or DS chunks, as well as when and how they are generated.
  - **TerrainMeshGenerator, HSMeshGenerator, VSMeshGenerator and DSMeshGenerator:** This Component is in charge of creating its own mesh, creating its vertices and their positions, and its triangles.

And the GameObjects that will have these Components attached to them:

- **Camera:** This GameObject will have the Camera and CameraControls Components attached to it.
- **Directional Light:** Light Component.
- **Player:** PlayerControls, MeshFilter and MeshRenderer Components.
- **Map Generator:** SurfaceGenerator Component.
- **Terrain Chunk, HS Chunk, VS Chunk** and **DS Chunk:** TerrainMeshGenerator, MeshFilter and MeshRenderer Components.

Additionally, a class was used to create the noise maps, although it is not a Component, since it is not attached to any GameObject and does not inherit from MonoBehaviour. This class is called **Noise**, and only features a static method that is called by any Component that wants to use a noise function.

Bear in mind these Components and GameObjects are only part of the final iteration of the project, the endlessly generated 3D world. Additional Components were used in the making of this project that are not part of the final version that will execute when hitting the Play button, but will be included in the source code, for experimenting or simply for future reference. For instance, these components:

- **CaveMapGenerator2D and CaveMapGenerator3D:** These Components were used to generate the maps for the caves in 2 and 3 dimensions, respectively.

- **CaveMeshGenerator2D and CaveMapGenerator3D:** These Components were used to generate the meshes for the caves in 3 dimensions, taking the map generated by CaveMapGenerator2D and CaveMapGenerator3D, respectively, in order to achieve this. CaveMeshGenerator2D generates a mesh like the one shown in Figure 3.12, and CaveMapGenerator3D uses the voxel approach seen in Figure 3.16.

- **BiomeMapGenerator:** This Component was used to generate a biome map and represent it visually, it's not actually used by the SurfaceGenerator Component. This just creates a biome map and represents it on a Plane GameObject using a texture created on runtime.

- **Renderer:** This Component is a generic renderer, it is attached to a Plane GameObject, and it is only used by the BiomeMapGenerator and PerlinNoiseTextureCreator in order to render a texture that represents a biome distribution map and a Perlin noise function visually, respectively.

- **PerlinNoiseTextureCreator:** This Component creates a noise texture to represent a noise function, in this case Perlin noise. Just like the BiomeMapGenerator, it uses a Plane GameObject to represent it via a texture.

- **FollowPlayerLight:** This Component is simply attached to the same GameObject as a point light, and its purpose is to follow the player's position so that they can see in the caves, since they are completely dark.

4.3 Project structure

Following the Object-Oriented Programming paradigm [18], Figures 4.1, 4.2 and 4.3 show the different class diagrams of the project [1]. Keep in mind that only the aforementioned Components (which are sort of the Unity equivalent of classes) will be shown, for the sake of simplicity.

[1]The base classes Unity implements are highlighted in red. These classes act as Components that are attached to the same GameObject as the Component they're connected to in the diagram.

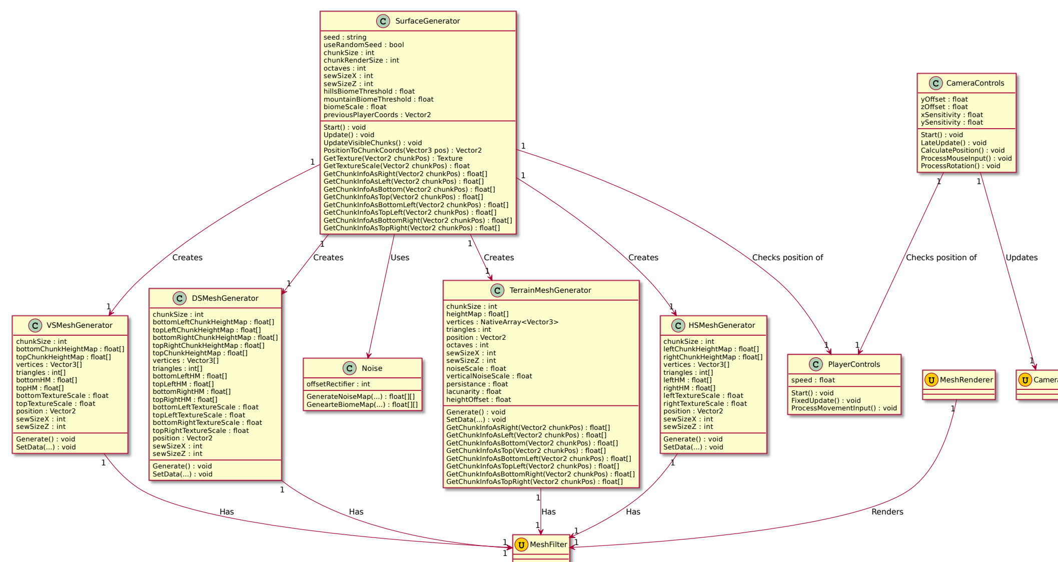

Figure 4.1: Class diagram for the main execution of the program. In essence, the process consists on: SurfaceGenerator checks if the player has changed positions, and in that case, it creates a number of TerrainMeshGenerator, HSMeshGenerator, VSMeshGenerator and DSMeshGenerator, which create the mesh and use a MeshRenderer to render it to the screen. Meanwhile, CameraControls updates the game camera so that it follows the player.

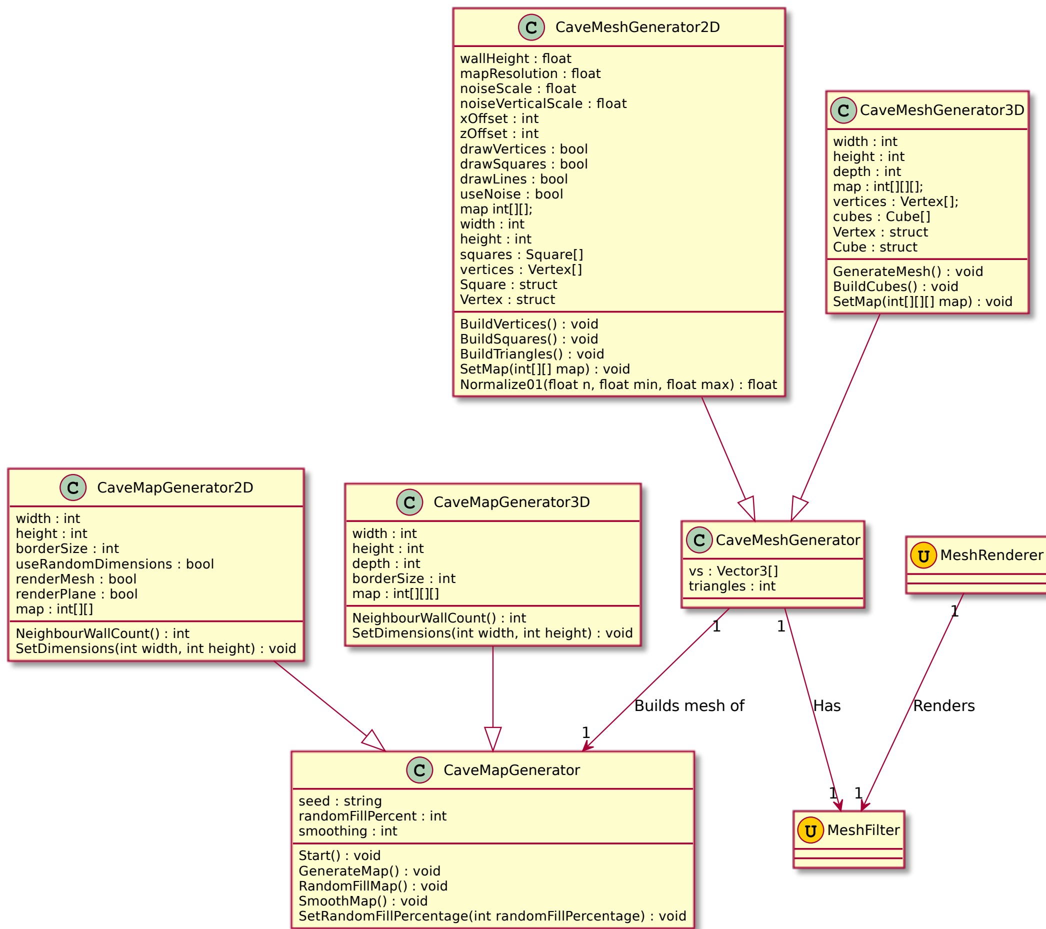

Figure 4.2: Class diagram for the execution of the cave generation part. This process consists on a CaveMapGenerator (which can be either 2D or 3D) creating a map which a CaveMeshGenerator will build a mesh out of later on.

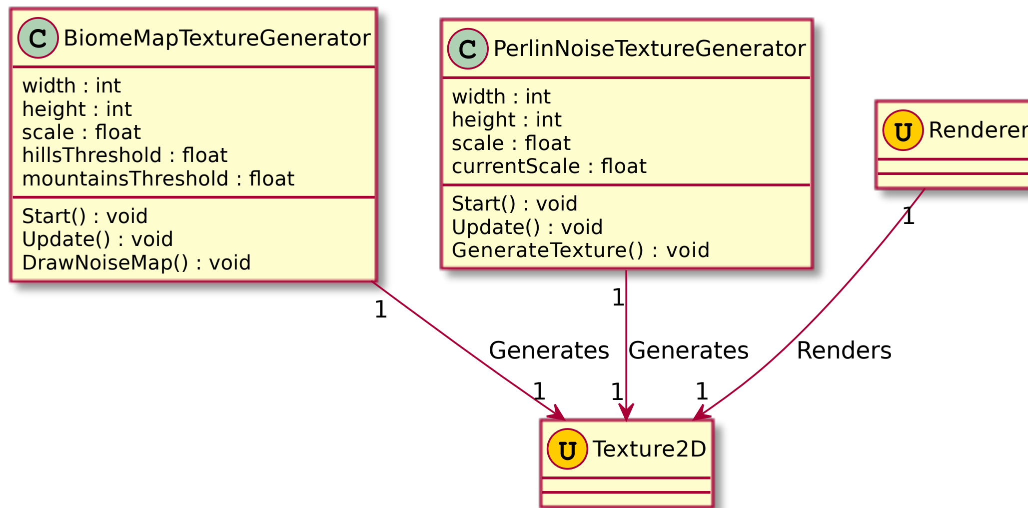

Figure 4.3: Class diagram for generating the Perlin noise and the biome map textures. These Components generate a Texture2D, which is later rendered by a Renderer Component..

Chapter 5

Simulations and Results

5.1 Simulations

In this section, some of the simulations made and the results obtained will be talked about. Specifically, we will talk about simulations with the mesh generation and how we think it could be improved, and we will also talk about the performance of the project, and also how it could be improved.

Marching Cubes

We mentioned previously that there exist several methods for building a 3D mesh out of a 3D heightmap, and that we implemented one based on voxels for the caves. One of these other methods is called **marching cubes** [14]. It yields a much more realistic result than the method we chose, and could be used for the surface generation as well, which would be more consistent with the surface terrain we generate, and thus, the integration with it and the transitions between it and the caves would be smoother.

This algorithm consists on defining the triangle topology of a mesh, and then constructing the corresponding triangles. To achieve this, we will take blocks of 8 of our heightmap positions. Then, for each block we will draw certain triangles depending on which of those heightmap positions are walls. More specifically, intermediate vertices will be created at the middle of the edges that connect every wall heightmap position to every other adjacent position. These intermediate vertices will be the actual vertices of the mesh we will render later. This might seem confusing at first, so let's take a look at Figure 5.1 to understand this better.

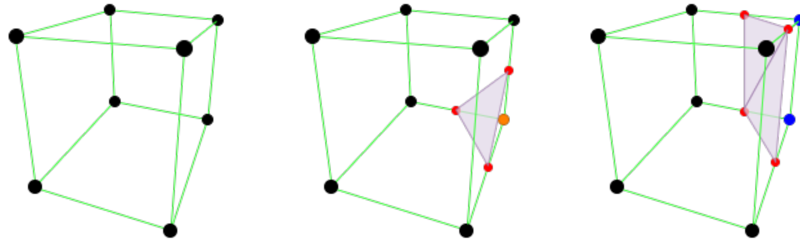

Figure 5.1: Three examples of the vertices and triangles that would be created for a given subset of 8 vertices. Each heightmap position which is not a wall is drawn in black, whereas the ones that are walls are painted blue. The intermediate pixels are shown in red, and the triangles that would result from it are painted purple.

It is important to mention that when creating the mesh that represents our heightmap, we won't take the heightmap positions as the mesh's vertices, but rather the intermediate vertices created (shown in red in Figure 5.1), and the mesh's triangles would be the triangles created out of these vertices.

For the implementation, we will assign each of the heightmap positions a vertex (local to that cube), which we will then multiply by its value raised to the power of 2, such that every possible combination of walls and non-walls is represented by a number. For instance, let's define them in the way we show in Figure 5.2.

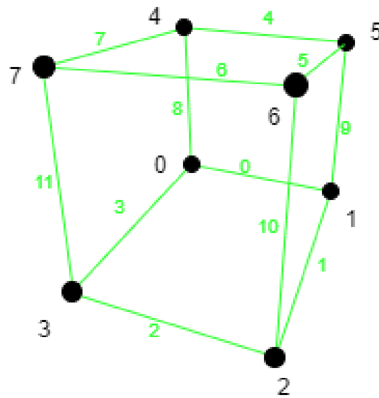

Figure 5.2: Definition of the vertices' and edges' indices (in black and green, respectively) of a cube. These indices are local to each cube, so each cube would have its own vertex indices from 0 to 7, and its own edge indices, numbered from 0 to 12.

Then, we will call a new number which we'll call the cube's index. This index will be calculated using formula 9.

$$c_i = \sum_{n=0}^{7} (v_n * 2^n) \tag{9}$$

where c_i is the cube's index, n is the vertex index, and v_n is the value of the vertex with that index (which will be 1 for walls and 0 otherwise). This way, we will have a different cube index for every possible combination of wall and non-wall vertices. To clarify, let's look at the examples from Figure 5.1.

- The left-most cube would have an index of 0, since none of its vertices are walls.
- For the middle cube, since only its vertex 1 is a wall, its cube index would be $1 * 2^1 = 2$.
- Lastly, since the right-most cube only has the vertices 6 and 7 as walls, its cube index would be $1 * 2^6 + 1 * 2^7 = 64 + 128 = 192$.

And finally, we will have a list of arrays containing the edges that the mesh triangles will intercept, called the **triangulation table**. As such, each of these arrays will have 12 positions, one for each of the edges of the cube. To get the one we want, we only have to look the position of that list that is equal to the cube's index. We can look at this as if it was a binary number, where every vertex is a bit. For instance, for the cube with index 0, we will look at the position 0 of the triangulation table. In this case, it is empty, since we don't have to draw any triangle. A better example is the cube with index 64, which, if we look up its entry on the triangulation table, we get 0, 1, 9, -1, -1, -1, -1, -1, -1, -1, -1, -1 - where -1 means no edge. This means that we will have to create the intermediate vertices that intercept edges 0, 1 and 9, and build a triangle that features those vertices in that order. If we check Figure 5.1, we can see that this is indeed correct.

Applying this technique to our project would result in something like what's shown in Figure 5.3.

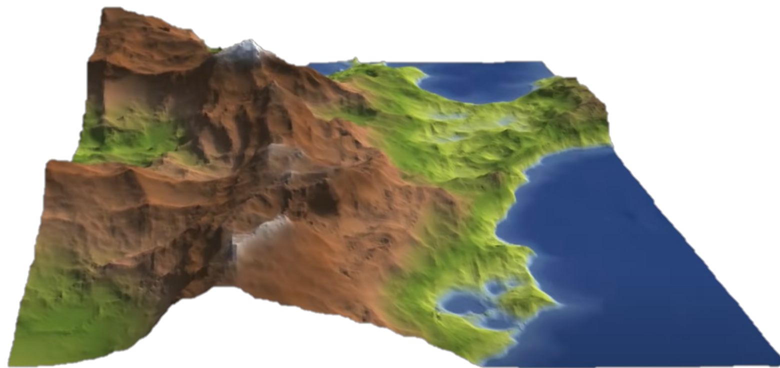

Figure 5.3: What our terrain surface would look like when using the marching cubes algorithm. Depending on how much we space each vertex, the mesh would look smoother or sharper.

Nonetheless, as we already explained on chapter 2, this algorithm has problems such as generating overhangs and floating islands such as the ones seen in Figure 5.4, which was not adequate for the project, since their physical feasibility is often times questionable, such as what's shown on Figure 5.4.

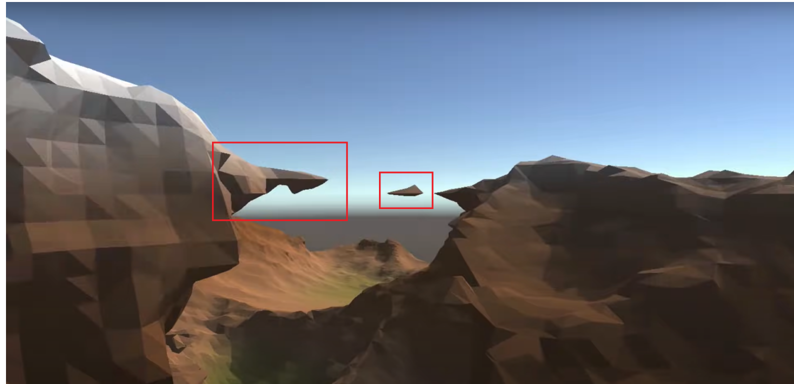

Figure 5.4: Unrealistic overhangs and floating islands generated by the marching cubes algorithm.

Multithreading

One of the biggest problems that we found when developing the project is performance. We implemented a multithreaded solution using a Unity feature called **Jobs** [6]. The Job system is a feature that was introduced recently by Unity as part of their new ECS system [20], which stands for Entity-Component-System. ECS is a new paradigm of programming that is focused on giving *performance by default*. This system changes from the traditional Object-Oriented Programming paradigm to something they call Data-Oriented Design. The ECS system is designed specifically to give massive performance gains by default, but, as we said, the way to program with it is drastically different than with their usual MonoBehaviours.

In order to parallelize our chunk generation we used a type of Job called IJobParallelFor. This specific Job carries out a piece of code the execution of which is split between the different threads. It works as a *for* loop, in that every thread will execute the same piece of code, and in that every thread will execute that piece of code with an index parameter, much like the index in a *for* loop. For instance, we could use this type of Job to parallelize a loop that just prints the index, so we would only need to override a function called *Execute* which takes a parameter index, and include in it the code that will be ran by the threads, in this case, just printing the index to the console. Then, when we call the function *Schedule* on that Job, the program will assign a number of indexes to be executed on each thread.

We implemented the IJobParallelFor VerticesJob in our TerrainMeshGenerator, HSMeshGenerator, VSMeshGenerator and DSMeshGenerator classes, where every chunk would create a Job that would calculate its vertices and schedule the appropriate number of indexes needed to complete it. For instance, with a chunk

size of 32 and a sewing size in the x axis of 3, the TerrainMeshGenerator class would schedule the Job with (32 * 32) indices, the HSMeshGenerator and the VSMeshGenerator would schedule it with (32 * 3) indices, and the DSMeshGenerator would schedule it with (3 * 3) indices. We split the calculations for the triangles on a different Job called TrianglesJob, which does the same as the VerticesJob, but with the chunk's mesh's triangles.

The calculation of the vertices' positions is the same as previously, since all we really did was parallelize the same *for* loop that was used to build the meshes, and split its execution among the threads. After implementing this we found a substantial increase in performance, where updating a row of chunks (with a chunk size of 128) took around 300ms, instead of the previous value around 600ms.

Nevertheless, as it's shown in Figure 5.5, when updating the chunks, the threads are idle most of the time, instead of doing calculations. We suspect this might be due to the fact that there are multiple Jobs being created and scheduled, and that has a considerable slow down attached to it.

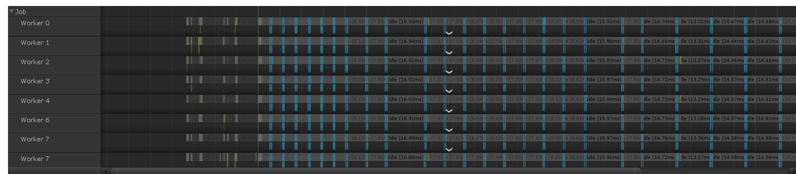

Figure 5.5: The time it takes for the Update method to complete, seen in Unity's Profiler window. As we can appreciate, the workers (the different threads) are idle a considerable amount of time (the blue regions represent the time periods when they are doing work), presumably from creating multiple Jobs.

The Job system is designed specifically to be used in tandem with the ECS system, although it can be adapted to be used with MonoBehaviours. Nonetheless, it has limitations, such as being unable to call any high-level Unity function, such as instantiating or destroying objects, such as meshes. For this reason, a more correct multithreading approach would have to be building all the meshes in a batch-like mode, where the SurfaceGenerator class would only schedule one Job, with $(chunkSize * chunkSize * chunksToRender)$, where *chunksToRender* would be how many chunks are rendered around the player.

5.2 Results

Finally, the results seen throughout this entire section's Figures were obtained at different iterations of the project.

Figure 5.6 shows a portion of the map generated completely with the hills biome. We can see how there are no gaps between the chunks, and that the textures are also sewn together (although, in this case, it's not clear since all the chunks have the same texture).

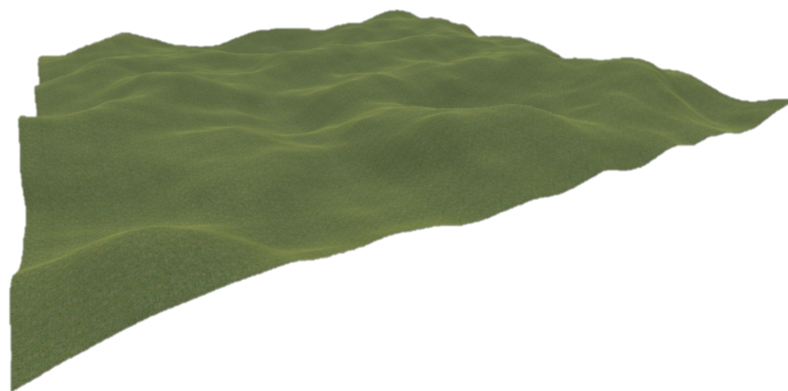

Figure 5.6: A portion of the hills biome.

Figure 5.7 shows the 3 biomes of plains, hills and mountains together. We can see the transitions between them.

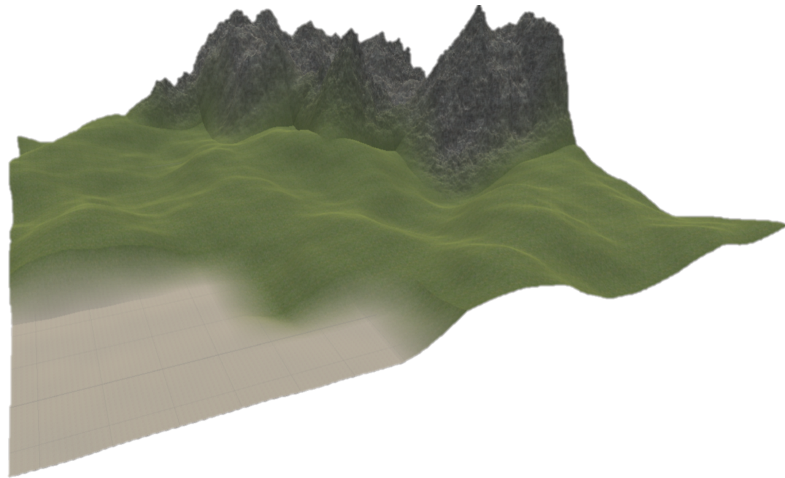

Figure 5.7: The biomes of hills, plains and mountains, and the transitions between them.

Figure 5.8 shows the transition between a portion of the hills biome and a portion of the plains biome. As we can see, the transition between them is quite seamless.

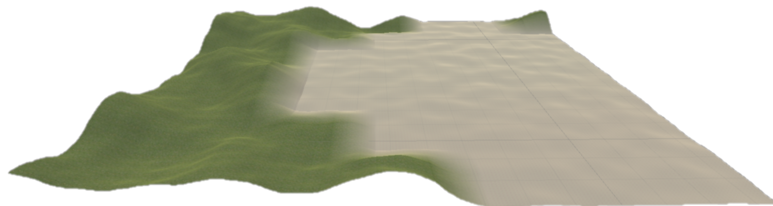

Figure 5.8: A portion of the hills biome transitioning into a plains biome.

After adding a 3-dimensional model, and some fog to conceal the end of the map, we obtained results such as those shown in Figures 5.9, 5.10 and 5.11.

We also decided to not interpolate the mountain biome's texture with other biomes, since, as we can see in Figure 5.7, the high verticality natural of mountains doesn't fit well with the texture interpolation. The way this is achieved is by, if any of the chunks that are going to be interpolated are a mountain chunk, then all of the sewing chunk's textures are set to the mountain texture.

Figure 5.9: A mountain range, displaying the model used for the player and the fog introduced in this part.

Figure 5.10: The reverting of the mountain texture interpolation between chunks. It makes sense in concept, but having square chunks of this size makes it look strange.

Figure 5.11: A figure displaying a wide variety of the biomes, with a new addition of the water biome. As we can see, the textures are only interpolated between non-mountain chunks. We can also see that the transition between the water and the plains biome is especially smooth.

Chapter 6

Conclusions and Future work

6.1 Conclusions

In this section, we will analyze the initial goals and explain the conclusions we reached.

- **Analyzing the most common PCG techniques.**

  We investigated commonly used techniques such as CAs, noise functions, and biomes. Having implemented all of these techniques, we reached the conclusion that, even though difficult, they proved to be extremely rewarding in terms of results, and they are indeed the future and a big research topic for a given area of game development. We have even seen how some of these methods don't have to be exclusively used for content generation, like the CA for the Game of Life.

- **Designing a game environment to include different PCG techniques.**

  Thanks to the Unity game engine, we were able to develop a scenario where the player can act freely and interact with the world. In this scenario, the player can move around and explore, and their actions will have an effect on what's being generated around him.

- **Analysis, design and development of cave-like environments using PCG techniques.**

  We dug a little deeper into different methods for the generation of the caves, mainly the CA method. We have studied how this method works and implemented it to generate both 2-dimensional and 3-dimensional caves, although both represented using 3-dimensional meshes. Even though they might not

always look like realistic caves, the results have been satisfactory when it comes to resembling what caves usually look like in video game culture. We have also seen several methods to build these caves' meshes, like simply having a mesh whose vertices are a certain height if they are a wall, or using a voxel-based approach. We have even seen how a different algorithm like the marching cubes would work, and the results it would yield.

- **Analysis, design and development of landscape using PCG techniques.**

  We have researched how noise functions work, and how they can be used for all sorts of content generation. From generating the terrain itself thanks to stacking noise functions, to creating a distribution map for the biomes. We have also seen how they could even be used for distributing and placing small objects like trees or rocks into our world, making it come even more alive.

  We have even seen how to sew different pieces of our terrain together, interpolating their heights and making extra bits that chain everything together seamlessly.

  We have also learned how to apply different materials and textures to our terrain pieces, how to apply these textures effectively to our terrain and get visually correct and satisfying results, and how to calculate the textures for the pieces of terrain that sew everything together, so that the transitions between chunks is also seamless in the artistic department.

- **Analysis, design and development of chunk generation techniques and its parallelization.**

  We have effectively proposed a way of generating the terrain in real time as the player moves around the world and explores it, splitting the terrain in chunks whose properties determine what their associated mesh will look like. As we have also seen, the generation of the world is virtually endless.

  We have also proposed a way of parallelizing the generation of these chunks in order to get a performance boost, even though it wasn't as big as expected. Nonetheless, this is a topic that if done well, could bring a massive improvement to the game.

- **Analysis, design and development of biome-based distribution of chunks.**

  We have also studied techniques specific to this project, like the distribution of biomes following a somewhat realistic approach. We have proposed several ways of distributing these biomes and reasoned why the chosen methods were indeed chosen over the other ones.

We have observed that the results obtained with said methods are satisfactory enough, and follow the original objective of generating an endless world, while at the same time, keeping a certain amount of realism and fidelity.

- **Design of player adaptability.**

  We have balanced many options for how the world should change based on the player's actions. Although admittedly, this part isn't the brightest of the project, we saw how even with such a limited playability we are indeed able to make the content we generate adapt to the player's actions. Had the project had more features, the possibilities would have been many more.

- **Validation of the final deployment**

  We have seen the execution of the program, and analyzed its memory and CPU usage, although contrary to the initial objective, it was decided midway that a final deployment would not be made, since the point of the project is to serve as a laboratory or a sandbox, where anyone can experiment. The project became more of a tool rather than a game, and thus, making a deployment for such a tool, having the Unity engine already functioning as such, would be redundant and unnecessary.

6.2 Future Work

- **Rendering method:** A small upgrade that could be done to the project would be, instead of generating the chunks around the player, to shoot a ray from the camera's position and render the chunks that would be in sight of the camera. This way, rendering the chunks that the player doesn't see is not necessary. This would also give a considerable boost in performance.

- **Adding more biomes:** Another change that would add more diversity would be adding more biomes to the world generation. This is a simple task, but one that takes time, since experimenting thoroughly with the noise parameters of the biomes is needed to achieve satisfying results.

- **Multiple textures per biome:** Another small detail that could add more to the aesthetic of the game would be to have multiple textures for each biome, for example, depending on the height. This way, mountains would be able to have snow at the top, and since we could also interpolate between the textures, it could gradually become darker and darker, simulating the snow melting away, which could give a nice touch visually.

- **Objects on the surface:** Finally, another aesthetic improvement would be to add simple objects on the surface, such as trees, rocks, or even simple animals. With our current knowledge this would be straightforward, since we could use yet another layer of noise to distribute these items throughout the world, and could give the world a feeling of not being empty.

Appendix A

Technical manual

A.1 Software versions

The following software was used, in their respective versions:

- **Unity:** version 2019.2.17f1
- **Microsoft Visual Studio Community:** version 16.10.1
- **Microsoft .NET Framework:** version 4.8.04084
- **Visual Studio Tools for Unity:** version 4.10.2.0

A.2 How to install

A decision to not make a standalone release for the game was made, mainly because the main point of the program is to be able to experiment oneself, rather than playing the little game the way we decided, so in order to know how to experiment with it, refer to the section A.3.

A.3 How to execute

In order to execute the program, we first need to download and extract the project folder from the zip file. Once extracted, we open the project with Unity (it is necessary that the installed Unity version is the exact same, otherwise, Unity will

not let the project open). Once the project is open, we will see a list of GameObjects in the scene on the left panel. These GameObjects are all explained in section 4.2.

In order to experiment with the terrain generation, for instance, we will need to click on the SurfaceGenerator GameObject on the left, which will then open the editor shown in Figure A.1.

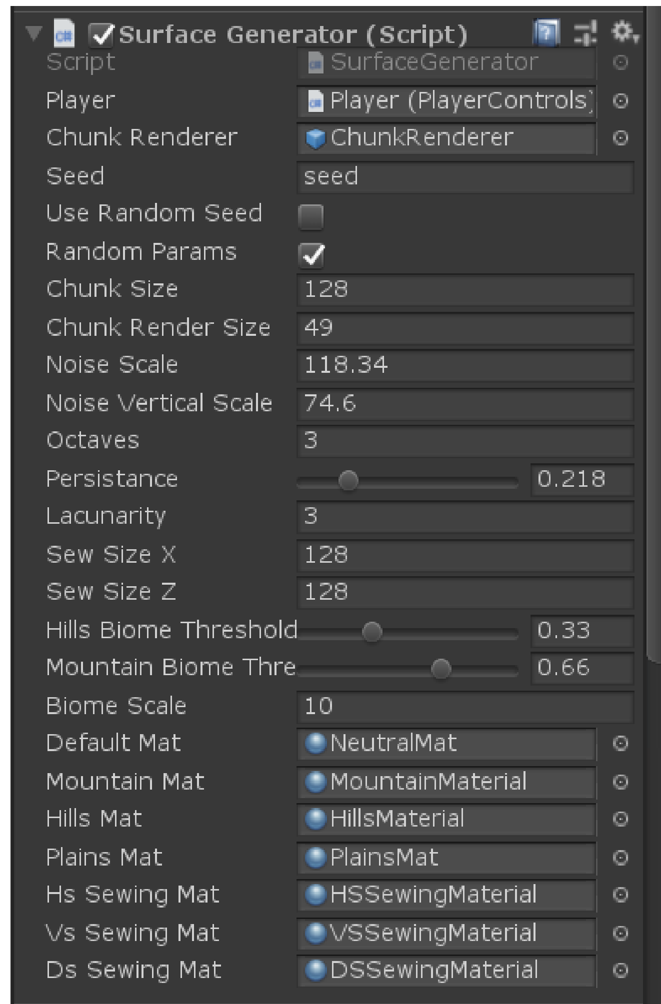

Figure A.1: The Unity editor when clicking on the SurfaceGenerator GameObject that is present in the scene.

We can see all the parameters that we can tweak using either numbers directly, or a slider for those parameters that are constrained to a range of values.

Once we have set the parameters that we want, we can hit the Play button at the top of the UI and the execution will begin [1].

The camera is a third person camera, which follows the player from behind, and a free look mode can be activated by holding the left mouse button and moving the mouse around. The movement is simple and follows a traditional W-A-S-D control scheme, where:

- **W:** move forward.
- **S:** move backwards.
- **A:** turn left.
- **D:** turn right.

[1] It is important to make sure the GameObjects we want to work with, like the SurfaceGenerator are *active*. In order to make a GameObject active or inactive is by ticking the box on the top of the editor when having that GameObject selected.

Bibliography

[1] *Beneath apple manor — Wikipedia, the free encyclopedia*, Online; last accessed 29-May-2021.

[2] *Biome*, Online; last accessed 13-June 2021.

[3] *Polygon mesh*, Online; last accessed 1-June-2021.

[4] *Voxel*, Online; last accessed 2-June-2021.

[5] *No man's sky*, 2016.

[6] *Unity - manual: C job system*, 2018, Online; last accessed 18-June-2021.

[7] John Conway, *The game of life*, Scientific American **223** (1970), no. 4, 4.

[8] Juncheng Cui, Yang-Wai Chow, and Minjie Zhang, *A voxel-based octree construction approach for procedural cave generation*, (2011).

[9] Jean-David Génevaux, Eric Galin, Eric Guérin, Adrien Peytavie, and Bedrich Benes, *Terrain generation using procedural models based on hydrology*, ACM Transactions on Graphics (TOG) **32** (2013), no. 4, 1–13.

[10] Tom Hatfield, *Rise of the roguelikes: A genre evolves*, 2013, Online; last accessed 29-May-2021.

[11] Houssam Hnaidi, Eric Guérin, Samir Akkouche, Adrien Peytavie, and Eric Galin, *Feature based terrain generation using diffusion equation*, Computer Graphics Forum, vol. 29, Wiley Online Library, 2010, pp. 2179–2186.

[12] LearnOpenGL, *Face culling*, Online; last accessed 1-June-2021.

[13] Nikole Leopold and Johannes Unterguggenberger, *Engine 186 gpu rasterization-based voxelizer documentation*, 2019.

[14] William E Lorensen and Harvey E Cline, *Marching cubes: A high resolution 3d surface construction algorithm*, ACM siggraph computer graphics **21** (1987), no. 4, 163–169.

[15] Oxford English Dictionary Second Edition on CD-ROM (v. 4.0), *Caves*, 2009.

[16] Teong Joo Ong, Ryan Saunders, John Keyser, and John J Leggett, *Terrain generation using genetic algorithms*, Proceedings of the 7th annual conference on Genetic and evolutionary computation, 2005, pp. 1463–1470.

[17] Ken Perlin, *An image synthesizer*, ACM Siggraph Computer Graphics **19** (1985), no. 3, 287–296.

[18] Tim Rentsch, *Object oriented programming*, ACM Sigplan Notices **17** (1982), no. 9, 51–57.

[19] Noor Shaker, Julian Togelius, and Mark J Nelson, *Procedural content generation in games*, Springer, 2016.

[20] Unity Technologies, *Entity component system*, 2018, Online; last accessed 18-June-2021.

[21] Unity, *Mathf.perlinnoise*, Online; last accessed 12-June 2021.

[22] Stephen Wolfram, *Statistical mechanics of cellular automata*, 1983.

[23] Intel Developer Zone, *Pixel and planar image formats*, Online; last accessed 6-June-2021.