Procedural Terrain Generation Using Voxels

Student: Stefan Petrov Kopanarov

Date: 30/04/2019

Student No: 160476206

Degree: BSc Computer Science with Industrial Placement (Game Engineering)

Supervisor: Dr Rich Davison

Word count (without references): 14 612

Abstract

This paper studies the techniques used for generating terrain procedurally using voxels. A number of techniques used in modern engines for terrain generation are investigated, implemented and compared against each other based on two main criteria: performance and visual impact. A tool has been created in the Unity game engine which can be used as a starting point in creating procedurally generated terrain for games. The tool contains an implementation of Perlin noise, Simplex noise, Fractal Brownian Motion, Marching cubes and Flood Fill algorithm, all of which will be explained in great detail in the paper.

The results of the study will be presented as well as suggestions for further work in extending the tool.

Declaration

“I declare that this dissertation represents my own work except where otherwise stated.”

Acknowledgements

I would like to give my special thanks to my supervisor Dr Rich Davison for always giving me extensive feedback on my dissertation’s progress and providing me with guidance during the project. I would also like to thank Dr Neil Speirs, Dr Graham Morgan and Dr Gary Ushaw, who were providing feedback during our group meetings.

Table of Contents

Table of Figures

Chapter 1: Introduction

This chapter will introduce the motivation behind the study, its aim and objectives as well as what the structure of this paper will be. There are some keywords in this short chapter that may sound unfamiliar (e.g. procedural generation, voxels, terrain) but all of these will be explained in the next chapter to provide a better understanding of the study for the reader.

1.1 Motivation

Nowadays, the most popular form of entertainment are video games [1]. With enormous streaming content platforms like YouTube and Twitch where thousands of people enjoy watching people share their gaming experience. It is becoming difficult for game companies to keep up with players which demand new and different non-repetitive experience that is difficult to deliver at the pace players expect. Games with no randomness provide the same experience each time they are played which gets players bored quickly due the games' predictability. One way to solve this issue is to hire more artists which can spend days and months creating different terrain – a very slow and expensive process. A better alternative would be to use procedural generation for the creation of some elements of the game. Procedural generation is a way to generate large amount of non-repetitive content that is "Infinite" (theoretically infinite, but limited by computing power). Thanks to procedural generation, the experience of a game can be made less predictable and more immersive and unique each time a player logs into a game's world. There are video games built entirely around the idea of giving the player a pseudo-random experience in order to allow the game to be endlessly replayed and keep players coming back to it over and over. This is an important aspect of video games moving forward in the industry and games which manage to provide it prove to be quite successful, e.g. Minecraft and No Man's Sky. This is the motivation for writing the paper and investigating ways in which terrain data for video games can be procedurally generated. Games are real-time simulations and as such, one big challenge for generating data procedurally is performance and the trade-offs of sacrificing performance for the sake of visual appearance which is a common challenge in video games – a challenge that has been faced throughout the project. The study itself has presented a good challenge for the author's own skills and has led to the necessary knowledge gain to implement procedural generation in their future projects.

1.2 Aim & Objectives

The aim of this study is to analyse and implement techniques used in procedural voxel terrain generation. In order to achieve the aim, a tool has been implemented that allows comparison of different techniques used in generating terrain. This tool can be used as part of a game's terrain generation toolkit by providing multiple techniques that best fit a game's requirements.

In order to achieve the aim, the study is divided in the following objectives:

1.2.1 Investigate static, dynamic or a combination of both techniques for procedural voxel terrain generation and choose one of the approaches.

The generated terrain can either be statically generated before the player enters the world (while the game is loading) or dynamically generated around the player as they move. It is also possible to use a combination of both static and dynamic by generating a large part of the terrain while the game is loading and then only generating zones that the player is close to. Due to the way the tool is implemented, a mixed generation can be achieved by adjusting some of the tool's settings. An algorithm called Recursive Flood Fill was implemented. An exploration and implementation of multithreaded programming was carried out.

1.2.2 Investigate methods of procedurally generating terrain data and pick two for further analysis, understanding and implementation.

Initially, the plan was to only look at Perlin noise and Simplex noise but due to their similarity and the resources available discovered late during the background research, it became possible to further on analyse two more aspects of terrain generation – Fractal Brownian Motion (a combination of layers of noise that creates a more realistic terrain) and Marching Cubes algorithm (used for extracting meshes from voxels). An investigation on volume ray casting as an alternative to marching cubes was carried out and will be briefly explained in the following chapters. However, due to time restriction, this technique was not implemented but the information from the investigation will be used in future implementation.

1.2.3 Implement the procedural terrain generation and render terrain.

This objective directly builds upon the previous two and is one of the two objectives related to the actual implementation. All the researched information was developed into a tool that contains two simulations: first one of them investigating static and dynamic procedural generation techniques; second one comparing the performance of Perlin noise and Simplex noise, using Fractal Brownian Motion as well implementing Marching Cubes to form complex shapes out of the initially given voxel data.

1.2.4 Optimise and evaluate the performance of rendering terrain using 2 different techniques.

- *Optimisation* of rendering is a very important aspect, especially if terrain is rendered in real-time. It is important to ensure that the rendering can be smooth and no computational power goes to waste which, during research, lead to the need of investigating different methods during the study. Recursive Flood Fill algorithm was chosen for optimising the computational power when dynamically generating terrain around the player as they move. However, a few different ways of dynamically creating the terrain were explored before implementing the Recursive Flood Fill algorithm, and those will be thoroughly explained.

- *Evaluation* of the performance was done as the tool was getting developed in order to compare the different techniques both for the terrain generation and the performance of the different noise techniques. A comparison was made between the generation time with the two different noises, memory usage of the generation techniques, number of triangles generated using the different noise implementations as well as frame rate and frame time of the game.

1.2.5 Gain knowledge in procedural content generation for future projects.

The techniques investigated and used in the study will contribute towards the author’s future work on the development of their own game which will be built entirely around the idea of delivering a unique experience for players every time they play the game.

All of the techniques mentioned in the objectives are implemented and explained in detail further in the paper in the Background and Evaluation sections.

1.3 Paper Structure

The paper is laid out as follows:

1.3.1 Introduction

The introduction will give an overview of the motivation behind the undertaken project, the aims and objectives of the project and the structure of this paper.

1.3.2 Background and Research

This chapter will give an overview of the way the research was done and provide a high level brief explanation on each of the algorithms and terminology used. All the key concepts like noise, voxels, procedural, algorithms, will be explained in a very reader-friendly manner. Some technical information in key areas will be presented to help the reader better understand the outcomes of the paper.

1.3.3 Implementation and Methodology

A chapter with explanation of the developed tool, the technologies and techniques used during the implementation. An overview of the development methodology, an in-depth discussion, understanding of which is key for grasping the implementation, details of implementation and reasoning behind decisions taken, showcase and explanation of the developed software. This chapter also explains why some technologies were chosen over others.

1.3.4 Results and Evaluation

This chapter will present the results of carrying out the project and a validation of the way the objectives were met. An evaluation of the performance of the different approaches will be presented.

1.3.5 Conclusion

The last chapter will reflect on the project and its success, potential areas of improvement and future work to be carried out. The chapter will, in addition, outline what has been learnt with the carrying out of the project.

1.3.6 References

References of sources used throughout the paper marked with [n] where n is a number.

1.4 Summary

Video games are here to stay, that is for certain. In this chapter we had a look at what the motivation for undertaking this project is, its aim and the main objectives that have been achieved throughout the research. We also gave a very brief introduction at the techniques the reader will be looking at in the next chapters.

Chapter 2: Background and Research

This chapter sheds the light upon all the terminology that was thrown around until now: voxels, terrain, procedural generation, noise and types of noise, noise algorithms, Recursive Flood Fill, Marching cubes, static and dynamic generation. It takes a look at the technology chosen for the implementation and the reasons for picking it.

2.1 Research

The research was done by studying original papers on the relevant subjects which were available on ACM DL [2] digital library, Google Scholar [3], GPU gems, research papers from well-known developers who have contributed to the subject area as well as many informal resources online.

2.1.1 Voxels

"A volumetric pixel (volume pixel or voxel) is the three-dimensional (3D) equivalent of a pixel and the tiniest distinguishable element of a 3D object. It is a volume element that represents a specific grid value in 3D space. "[4]

Voxel is the smallest unit in a 3D space that carries information, just like their representatives in the 2D world – the pixels. People often misinterpret that voxels are cubes, as voxels are often presented like a stack of cubes, but it is important to understand that this is an incorrect and limiting interpretation of voxels. A voxel is a single data point in a 3D array, and not a volume. Yes, voxels can hold cube data, but in reality they can contain any shape.

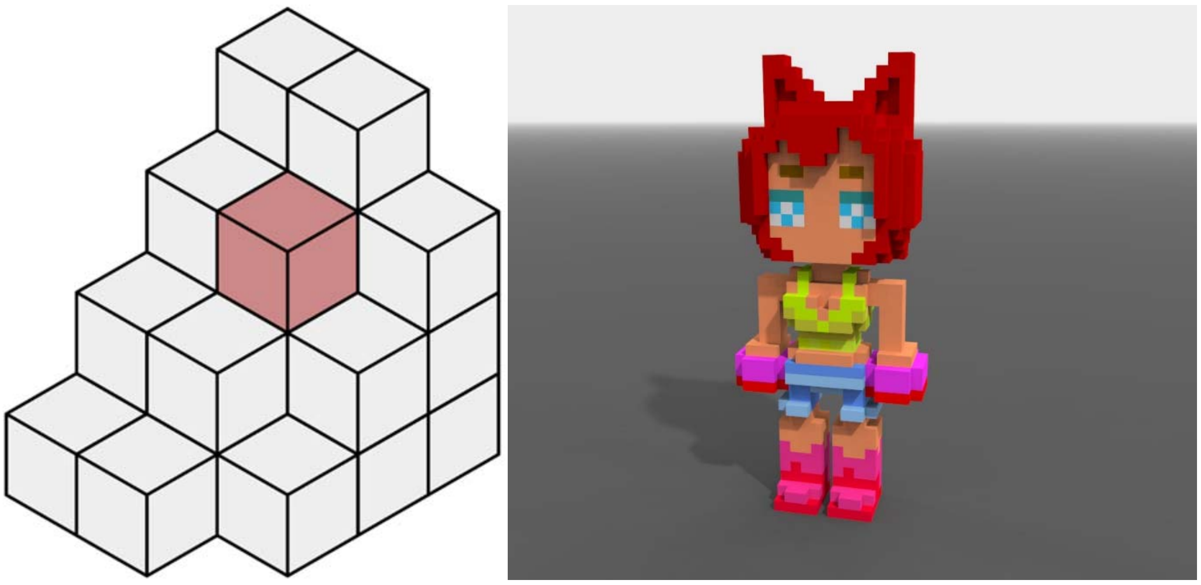


Figure 1: Voxels-based objects

On Figure 1: Left: A representation of voxels holding cube data, which is the most common interpretation of voxels [5]. Right: A 3D model of a female character made from voxels containing cube data [6].

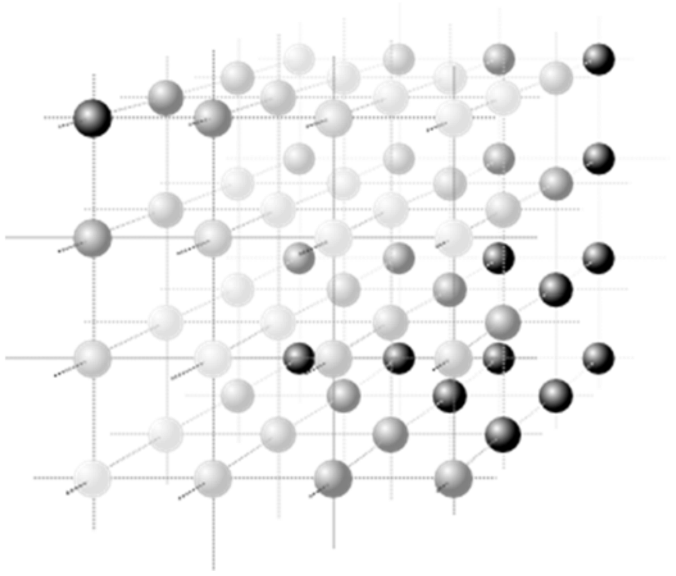
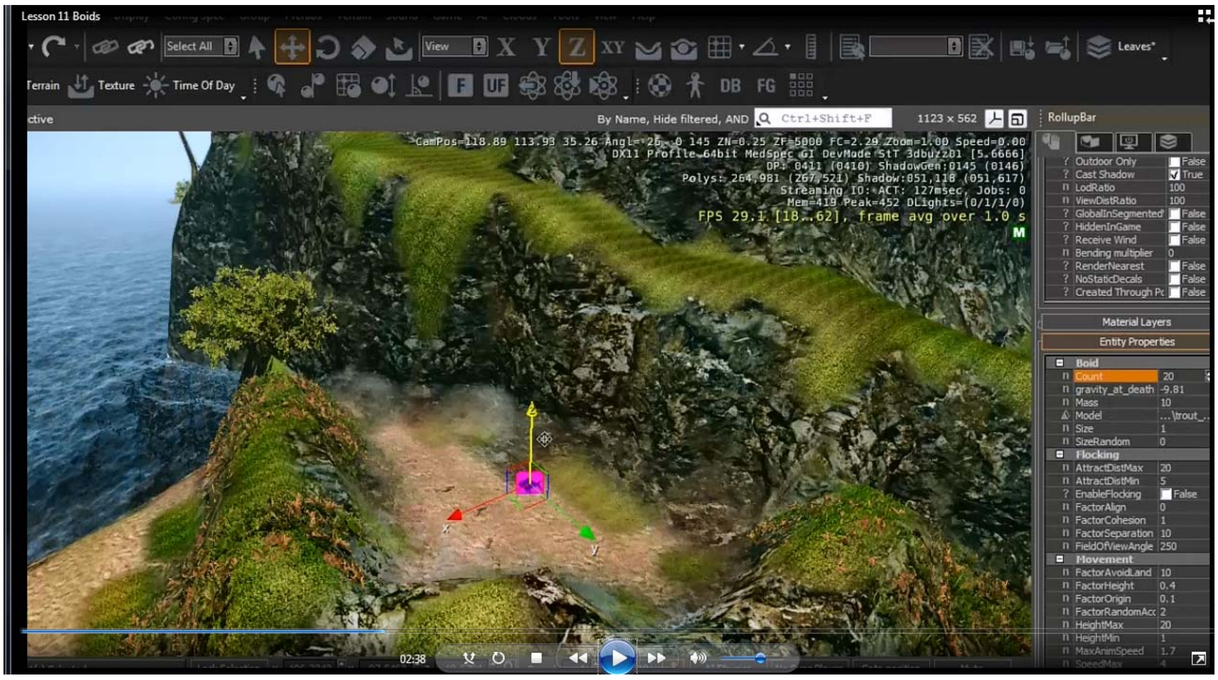


Figure 2: A voxel grid with voxels containing colour values [7]

On Figure 2, Illustration of a voxel grid, each containing a colour value - it is important for the reader to understand that voxels are not simply cubes as it will make it easier to understand how things like the Marching cubes algorithm work later on. This grid contains discrete values, meaning no elements can go in between these small circles, representing voxel coordinates on the grid. Because voxels are commonly misinterpreted as blocks, they are often considered very limited and only useful for blocky terrain, but a very good example of that not being the case is CryEngine's system that combines voxels and heightmaps [8] to generate its realistic terrain as can be seen in the top image of Figure 3. Another example is Unreal Engine's use of voxels for particle effects in FireFighting Simulator. [9]



```
CamPos=118.89 113.93 35.26 Angl= 26  0 145 ZN=0.25 ZF=5000 FC=2.29 Zoom=1.00 Speed=0.00
                 DX11 Profile 64bit MedSpec GI DevMode StT 3dbuzz201 [5.6666]
                                      DP: 0411 (0410) ShadowGen:0145 (0146)
                         Polys: 264,981 (267,521) Shadow:051,118 (051,617)
                                     Streaming IO: ACT: 127msec, Jobs: 0
                                       Mem=419 Peak=452 DLights=(0/1/1/0)
                                      FPS 29.1 [18..62], frame avg over 1.0 s
```

Outdoor Only	False
Cast Shadow	✓ True
LodRatio	100
ViewDistRatio	100
GlobalInSegmented	False
HiddenInGame	False
Receive Wind	False
Bending multiplier	0
RenderNearest	False
NoStaticDecals	False
Created Through Pc	False

Material Layers

Entity Properties

Boid	
Count	20
gravity_at_death	-9.81
Mass	10
Model	...\trout_..
Size	1
SizeRandom	0
Flocking	
AttractDistMax	20
AttractDistMin	5
EnableFlocking	False
FactorAlign	0
FactorCohesion	1
FactorSeparation	10
FieldOfViewAngle	250
Movement	
FactorAvoidLand	10
FactorHeight	0.4
FactorOrigin	0.1
FactorRandomAcc	2
HeightMax	20
HeightMin	1
MaxAnimSpeed	1.7
SpeedMax	4

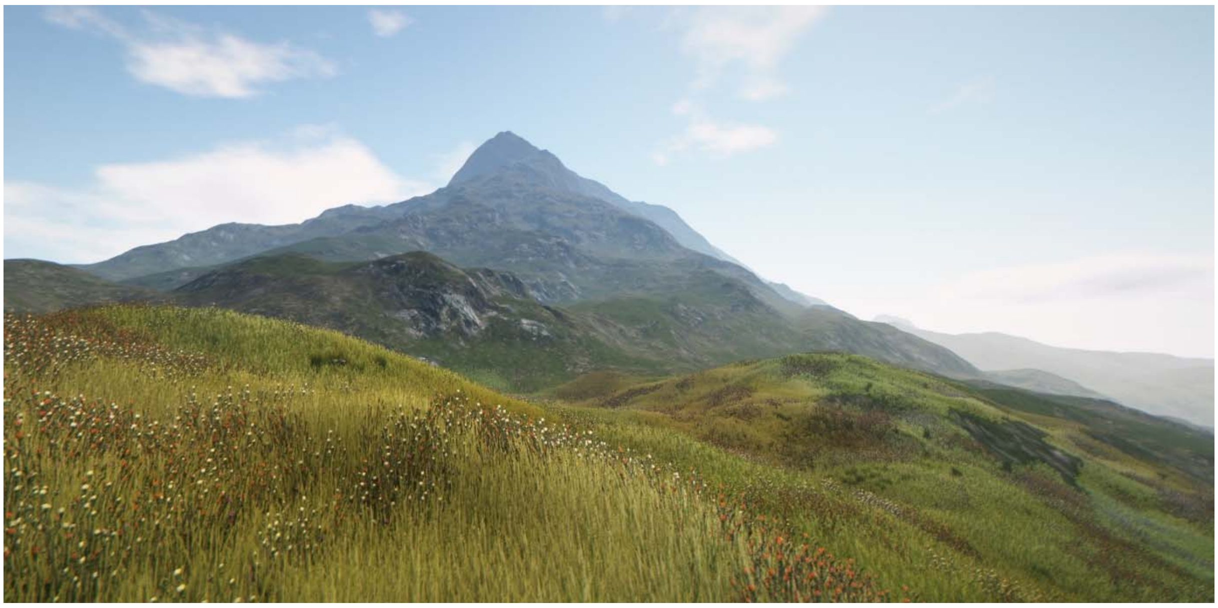


Figure 3: CryEngine3's system and a realistic terrain

On Figure 3: Top: CryEngine3’s system that combines voxels and heightmaps to generate realistically looking terrain. Bottom: An example of terrain showing a grassy and mountain area. [10]

2.1.2 Terrain and biomes

Terrain is the surface part of a game on which the player walks while playing the game. Biomes are the different types of flora and fauna combinations that occupy the surface of our planet. When generating terrain procedurally, a game can decide based on factors like sea level height, humidity, amplitude and latitude what kind of flora and fauna should be generated for a specific area. This study does not take biomes into consideration and only studies the creation of naturally-looking shape of the terrain. However, the way that the tool is written, it is aware of the notion of air and ground, which allows for it to easily be extended to take many biomes into consideration and will be extended as part of the future implementation of a video game.

2.1.3 Procedural generation

“In video games, it (procedural generation) is used to automatically create large amounts of content in a game. Advantages of procedural generation include smaller file sizes, larger amounts of content, and randomness for less predictable gameplay.”[11]

As mentioned before, due to the large popularity of video games, developers struggle to keep up with creating content as quick as customers demand it. Using procedural generation in games allows the games themselves to create large amount of their content which helps a developer cut both the cost of artists and time spent for development.

Figure 4 shows a great example of how No Man’s Sky game uses procedural generation to pick from different pools containing body parts and use them in various colour combinations to automatically end up with quite diverse creature models for the game. This same technique is applied throughout the entire game which practically provides an infinite amount of combinations of planets to explore - all created via procedural generation.

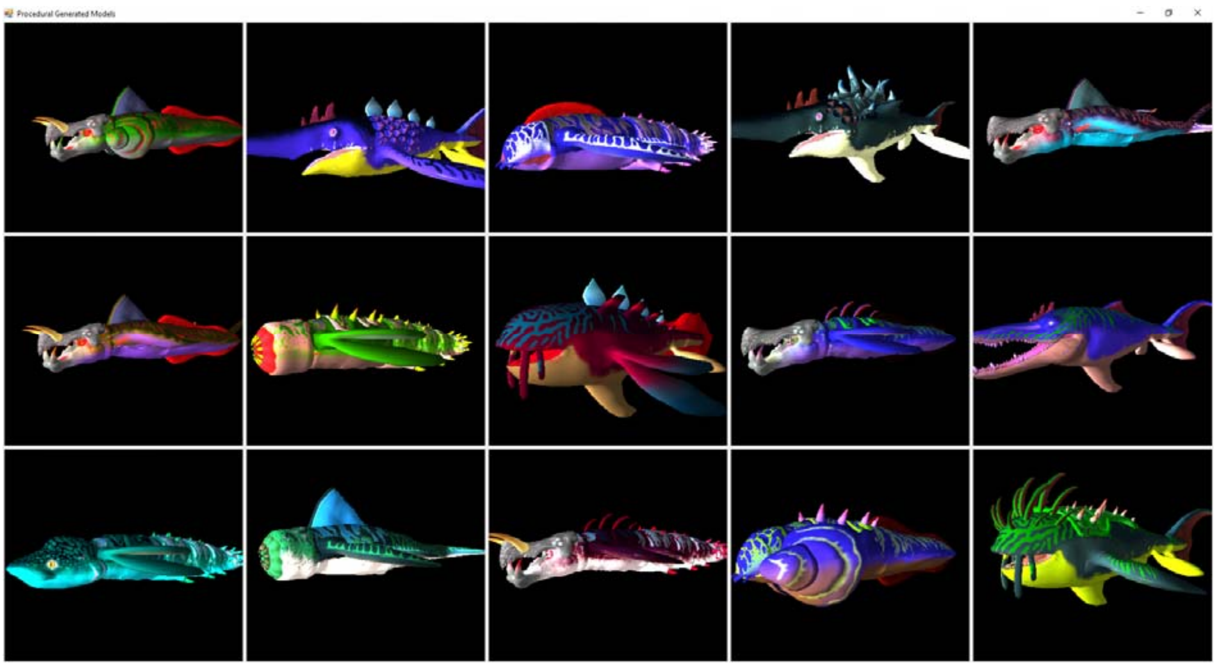


Figure 4: Procedural generation used in creating creature models in the popular game No Man's Sky [12]

2.1.4 Static and Dynamic Procedural Generation

When talking about procedural generation, static and dynamic generation should be considered.

Static is the idea that the content is created before a player enters the game itself. Imagine that the entire world gets created while the game is loading and once the game loads, there is no more world generation occuring.

Dynamic is the opposite of static, where data for the game gets generated at run-time. In terms of terrain, that would mean that as the player moves, the terrain around them will get created and destroyed. The easiest concept is to think that the game use the player's current position and a radius around them to determine where to generate new data and where data can be destroyed. Another way of implementing this could be by generating a random map layout for a dungeon or level each time a player enters it. The dungeons in Blizzard's franchise Diablo are laid out in such manner.

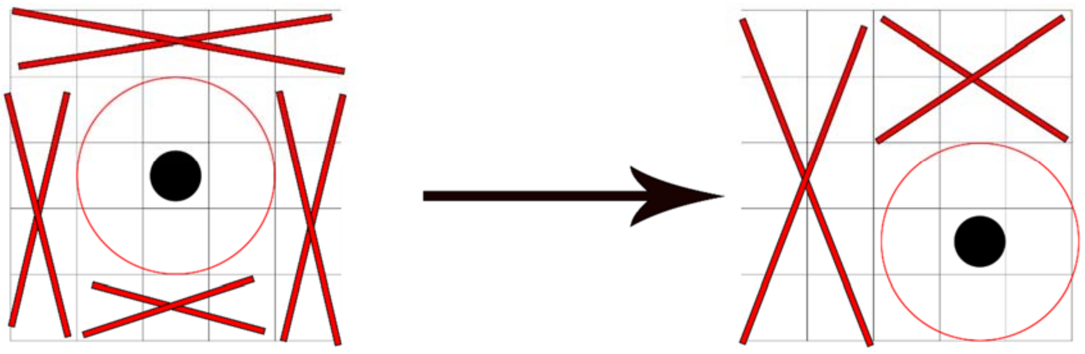


Figure 5: Dynamic generation around player

On Figure 5: A simple drawing that shows how the player (black circle) moves around and the terrain outside a certain radius (red circle) around them gets destroyed (blocks crossed with red lines).

Pros and Cons of Static and Dynamic Generation [13]

Static generation allows for picking the most visually pleasing results and putting only them in the game. The world can be quite complex and diverse, as the complexity and speed of generation does not affect the player's experience due to it being generated before the game is played. External software can be used and artists given vast freedom when creating custom assets for the game with no repetition at all.

Some disadvantages of static generation, however, are the size of data needed for the game to operate, the cost of creation and the limits to time and amount of art that artists can produce that can go into a game.

On the other hand, dynamic generation's pros are the infinite variations of content for a game that can be created, which can lead to opportunity for a lot of replayability and variating experience each time a game is played. In addition, dynamic worlds are not data-heavy as they are generated on the fly and are not restricted by artists' time and cost, due to content being generated by the game.

Dynamic generation also has its cons – it can be quite complex to implement and the randomness of a game experience depends on how well that complex world gets made. The complexity of the world is controlled by the speed and memory size of a player's machine which can lead to some players having to wait longer times while playing – a factor that may ruin one's experience if they are using an older computer. Moreover, due to the content being generated by a machine, sometimes odd behavior can occur which could be displeasing to the eye and ruin the experience of a player (see Figure 6)

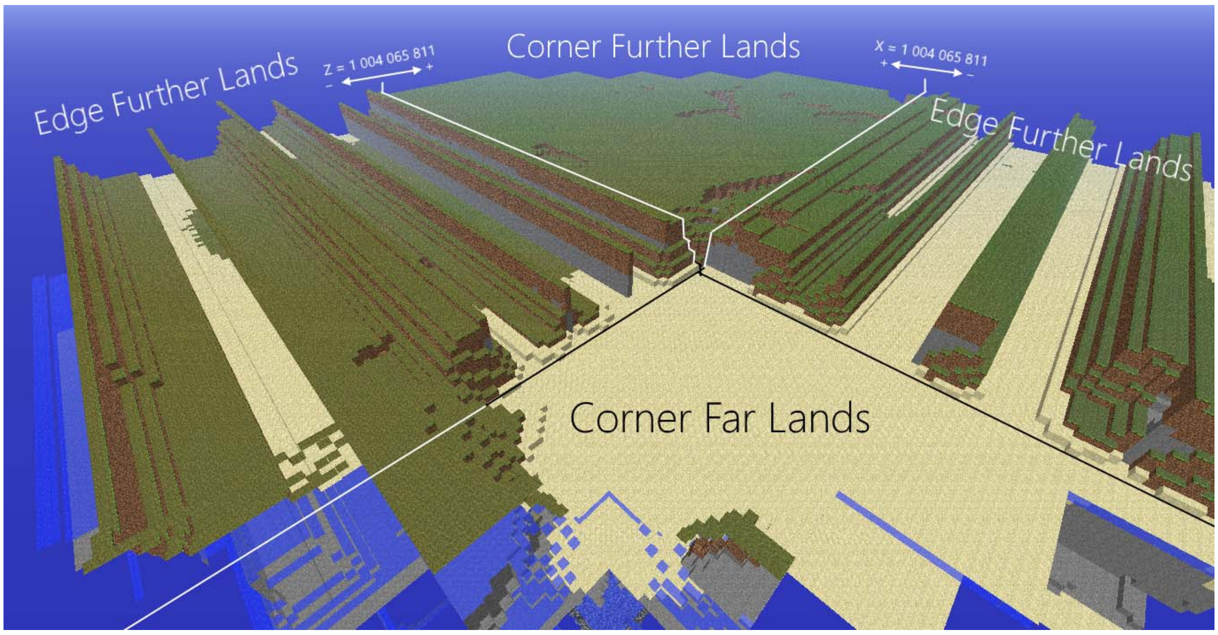


Figure 6: An example of the odd behaviour around edge of the world in Minecraft [14]

A note on "infinity": In Figure 6, it can be seen how the world starts to get glitches and unpredictable behaviour when it is very far from where the player started. This is caused due to floating point precision errors in calculations. Thus, when talking about infinity with dynamic worlds, it should be noted that "infinite" worlds are still restricted by the computational power of modern machines. The players can experience all sorts of visual glitches, frame drops and other unexpected behavior. Note that in the case of Minecraft this is quite far away from the starting area (±12,550,821–±1,004,065,920 blocks away) [14].

The different ways of terrain generation will be compared against each other and an evaluation will be held, the results of which will be talked about during the Implementation, Results and Evaluation chapters.

Recursive Flood Fill Algorithm (stack-based recursive implementation)

This algorithm is used to find out areas connected to each other in a grid of cells by starting from a certain position on a 2D grid and recursively marching in all four directions. The advantage of this recursive algorithm is that it doesn't scan diagonally, because each new square has to check its own four directions (see Figure 7). When a square sees that a neighbour has already been covered, it stops checking in that direction. Such is the case with the 4 new squares checking the initial one and stopping after that (thus having to check only in 3 directions). This algorithm is widely used in games like Go and Minesweeper to check for neighbouring stones/mines. It is also used in the "Bucket fill" tool in image-editing software. The process of reaching the implementation of the Recursive Flood Fill algorithm and its use in the dynamic generation will be discussed in greater details in the following chapters.

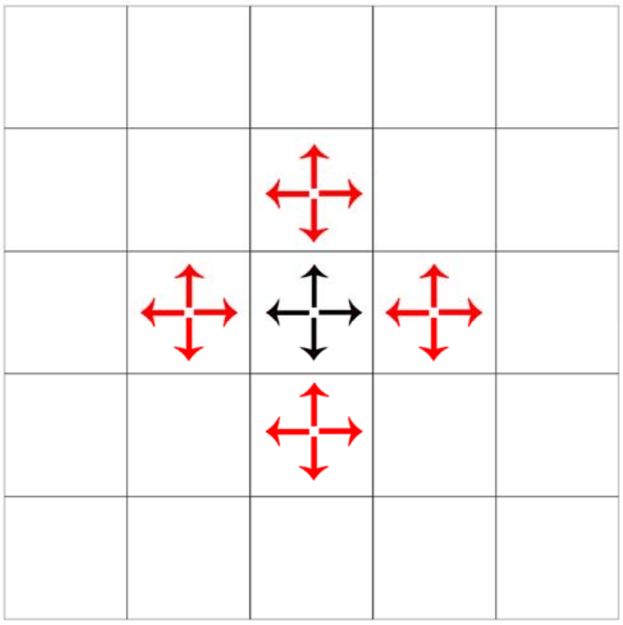
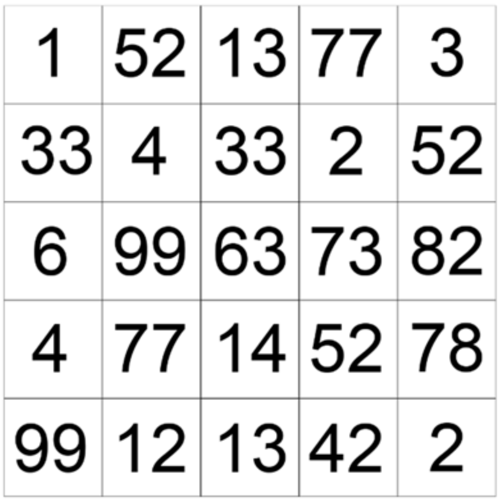


Figure 7: Visualisation of Recursive Flood Fill algorithm.

Figure 7 shows how the algorithm initially starts with the central square (black arrows), checks in its four directions and recursively does so in each of the four squares that follow.

2.1.5 Noise

If one did not know anything about procedural generation, but wanted to create terrain heights that can be used to represent the height of mountains, the steepness of hills and the depth of valleys, an initial brute force solution might be to think of going through each point of the map and adding a random number for that map position. Then each number can represent a certain level of the height with higher numbers meaning mountains and lower numbers meaning valleys.



1	52	13	77	3
33	4	33	2	52
6	99	63	73	82
4	77	14	52	78
99	12	13	42	2

Figure 8: Random Number Generated grid of numbers to represent heights

However, here are a couple of issues with randomly generating numbers:

"…two points on the surface of a same object which are far apart can look very different. In other words local changes are gradual, while global changes can be large. RNG (random number generators) do not have this property. Every time we call them they return numbers which are not related (uncorrelated) to each other. Therefore calling this function is likely to produce two very different numbers. This would be unsuitable for introducing a slight variation to the visual appearance of two points that are spatially close." [15]

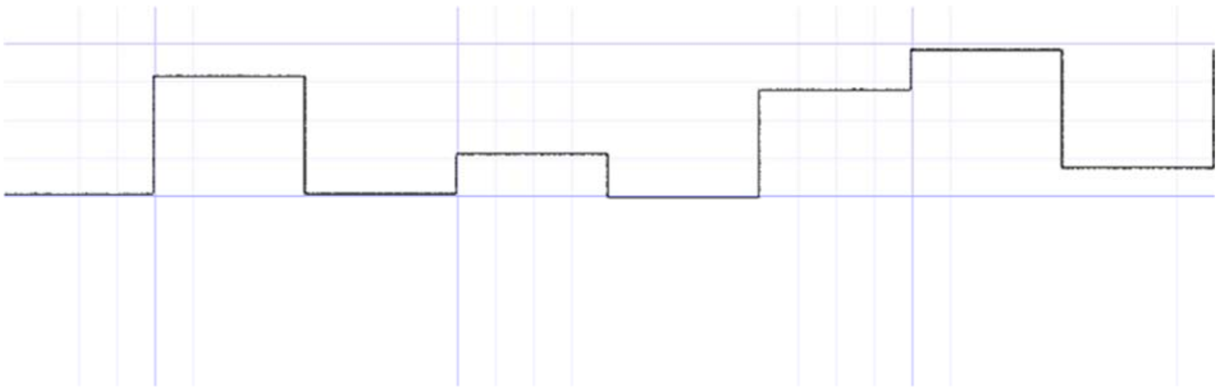


Figure 9: A wave-like representation of the steep jumps generated by random numbers similar to those on Figure 8. [16]

One problem with simply calling a random number generator function is that one cannot control the sequential relationship between the called numbers. This means that the random number generator does not consider the neighbouring numbers and it can come up with numbers that are way too far apart (e.g. 1 and 72 out of 100). This will generate a very shallow area followed by a very high area. But in real life, the patterns that can be observed have some level of smoothness and are not as random and drastically changing.

Another issue with random number generation is that if it was used for map generation, there would have to be enormous tables that store the data. A simple example is storing the data for a small 1000x1000 blocks grid terrain data would already require a table with 1 000 000 entries. This is quickly proving to be inefficient.

Enter noise!

There are two common types of noise used for procedural generation – value noise and gradient noise. This paper will focus on the gradient noise only as it acts as the noises studied here (Perlin and Simplex noise both build upon the gradient noise). On gradient noise:

"*This method consists of a creation of a lattice of random (or typically pseudorandom) gradients, dot products of which are then interpolated to obtain values in between the lattices.* " [17]

Noise also starts off with the idea of having a grid, containing numbers. However, noise allows to get rid of the idea of having to store millions of number entries which makes noise an extremely cheap and only requires input of the random numbers from the grid into a function. Technically, using noise is slower than texture sampling because it requires the execution of quite a few math operations, but its strength lies in the fact that it can create large variation and it does not require storing large texture data that must be constantly loaded and unloaded from memory:

“The noise function requires the execution of quite a few math operations (even if they are simple there's quite a few of them) while texture mapping only requires access to the pixels of a texture (an image file) that is loaded in memory.” [15]

Noise is great because it can work in many dimensions 1D, 2D, 3D, 4D… nD and is also deterministic - it will always produce the exact same result if it is given the same input - a great feature and allows for the same noise to be easily re-created for debugging or when used as a terrain generation tool.

Perlin Noise [38]

Perlin noise is named after its creator Ken Perlin who invented it back in 1982 for the motion picture Tron. He later won an Academy Award for Technical Achievement for this algorithm. It is one of the most-widely used techniques for using noise when generating content for movies and games.

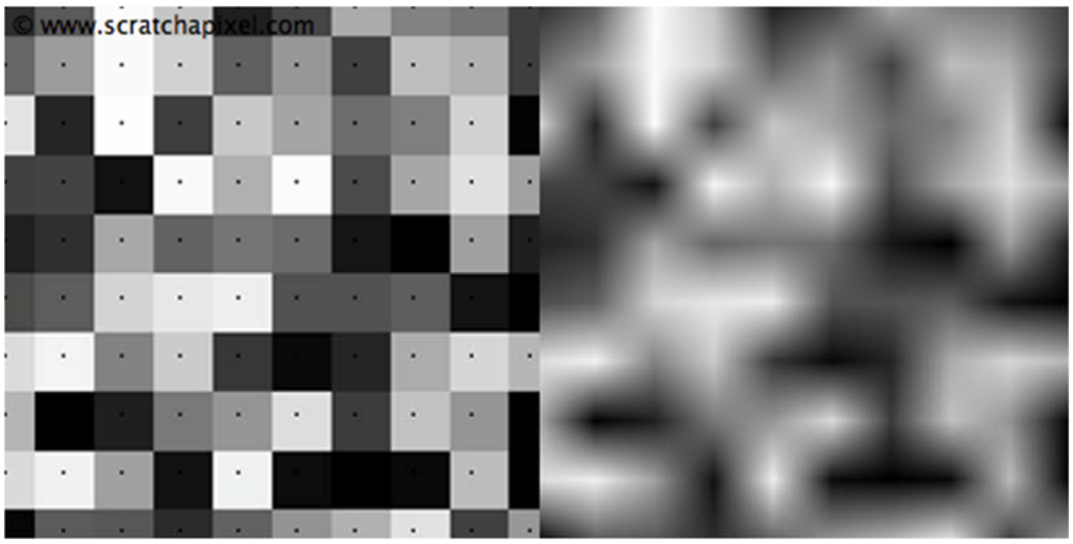


Figure 10: Left: Random numbers on a 10x10 grid. Right: Linearly interpolated scaled version of left image to end up with gradient values. [15]

Perlin noise also begins with a grid of random numbers, but it uses some clever mathematics and does two things that make it possible to achieve great effects at a very efficient performance level. The random values get linearly interpolated and then also get smoothed to further blur the random values and achieve a more curve-looking result.

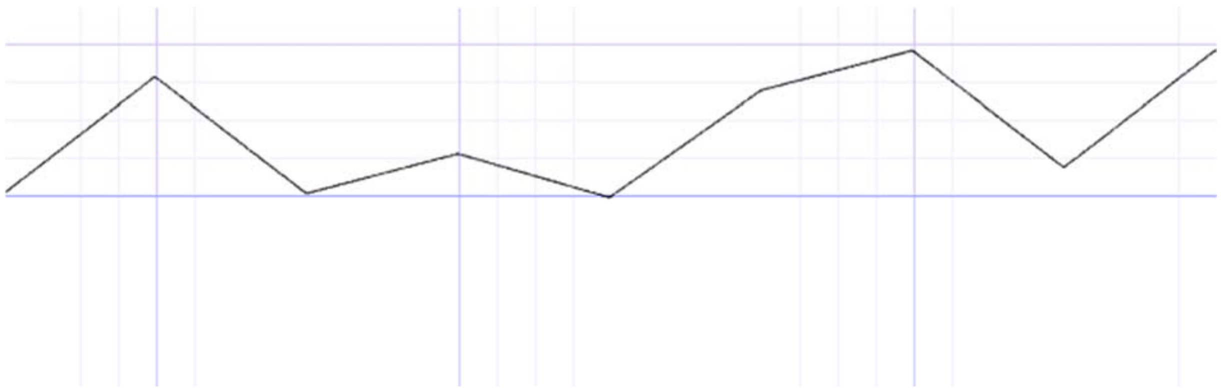


Figure 11: The result of the random values that already look much steadier (thanks to linear interpolation) and form a rough wave-like shape. [15]

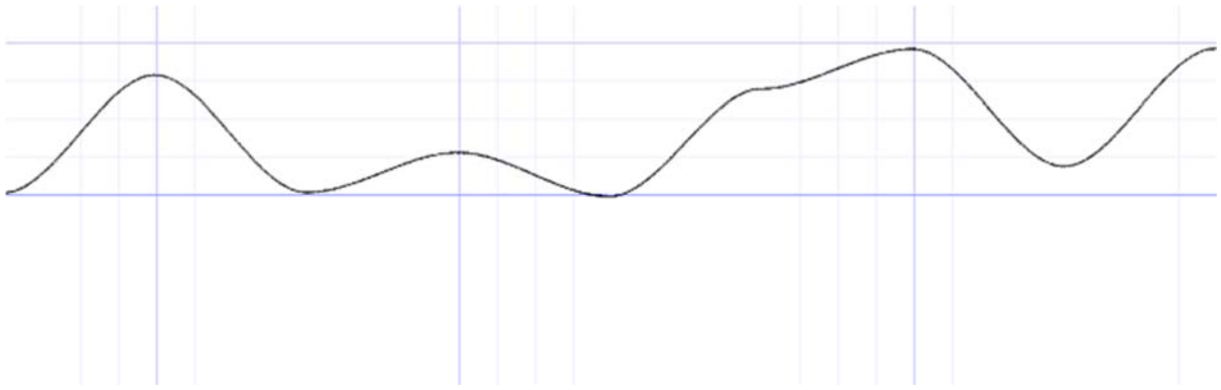


Figure 12: The result of the smoothing function of Perlin noise that makes for good-looking terrain silhouette.

Compared to Figure 9, Figure 12 looks much more like a landscape. Recall that the most important property of Perlin noise is its pseudo-randomness and determinism, meaning the exact same result can be produced for a known input. In addition, noise also has properties like frequency and amplitude which can be tuned to achieve steeper/steadier terrain, depending on the needs of the project it is used in.

Improved Noise [18]

Around 2001, Ken Perlin figured out that he could improve his own Perlin noise by adjusting the smoothness formula to achieve even nicer S-looking curves, in addition to some other minor changes. The improved noise version of the classical Perlin noise is the most widely used version of noise nowadays for two reasons – it performs better than Classic noise and it is not patented, as Simplex noise is. [18] The Improved noise is used instead of classic noise in this paper's tool.

Simplex Noise [39]

In 2002 Ken Perlin had seen some performance flaws in his classic noise and came up with a different algorithm that performs much better, especially in higher dimensions due to the way it is implemented. While classic Perlin noise uses grids that look like squares, the Simplex noise uses something called simplices (n-dimensional triangles).

"The advantages of simplex noise over Perlin noise:

- Simplex noise has a lower computational complexity and requires fewer multiplications.

- Simplex noise scales to higher dimensions (4D, 5D) with much less computational cost: the complexity is O(n^2) for n dimensions instead of the O(n2^n) of classic noise.
- Simplex noise is easy to implement in hardware." [19]

Simplex noise was used in the paper to compare against Perlin noise's performance and visual outcome.

Fractal Brownian Motion

Technically, Fractal Brownian Motion (FBM) is not a type of noise but a technique that involves combining different layers of noise. Each of these layers contain variations of amplitude and frequency. It adds some properties like gain (controlling amplitude), lacunarity (controlling frequency) and octaves (octaves are layers of noise, in which every next layer has double the frequency of the previous layer).

2.1.6 Marching Cubes

"Marching Cubes is a divide and conquer algorithm that creates triangle models of constant density surfaces from 3D medical data." [20]

What this algorithm does is walk through a field of voxels and based on a certain value that it's given, called isosurface, extract the terrain data to form a mesh that looks very realistic and smooth, rather than made of cubes.

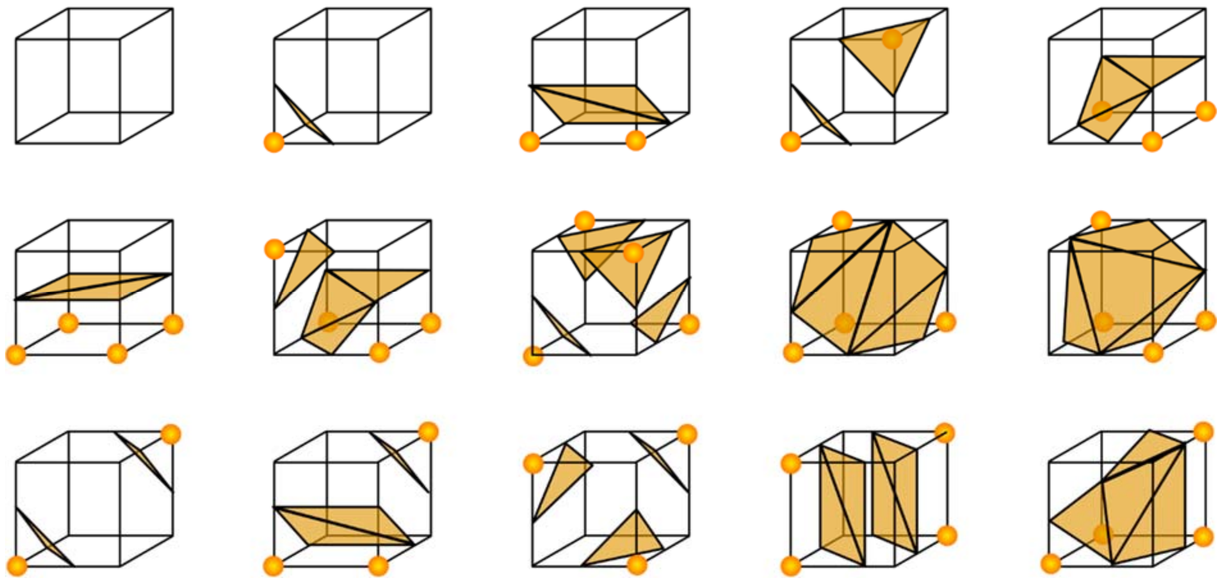


Figure 13: The originally published 15 cube configurations. [30]

2.2 Summary

There are many aspects to consider when talking about procedurally generated content for video games. Focusing on only one aspect of it – the terrain generation – does not make it any easier. On one hand, the ways to construct the terrain and make it look non-repetitive, natural and appealing is an enormous topic to explore on its own due to the amount types of noise and noise combinations available with all their adjustment and combination elements. On the other hand, the efficiency of generating and storing the terrain data is also a topic that can have entire paper written on it.

Chapter 3: Implementation and Methodology

This chapter gives a brief overview of the development strategy, the available technologies considered and reasoning behind which technology was chosen. The work undertaken in the project and the prototype software developed as a result of the studies will be discussed.

3.1 Development strategy

The topic of this study is very big which allowed for a lot of flexibility when deciding how much of a topic to explore and whether to move to a new topic or go deeper into the currently explored one. The purpose of this project was to explore two main topics – on one hand storing, loading, deleting and rendering data, on the other hand – making that data produce appealing and well-performing results.

It was decided to take an iterative approach, instead of a waterfall one, as the two topics could be explored almost in parallel. Taking the iterative approach allowed for mitigating risks like running out of time to produce a well-rounded tool that can be re-used at a later stage, not being able to implement certain aspects of the project or being restricted by the chosen technology. Having the opportunity to meet the supervisor at any given time, as well as having planned meetings with updates was a great way to see how far the project has gone and what areas needed to be further looked at.

One example of the iterative approach being helpful was picking up the research in volume ray casting which was picked at a stage that did not allow for implementation due to time restriction. However, there was enough time to research into it and see it as an alternative way of rendering and handling the terrain data for a future development. Because no other part of the implementation was dependent on the volume ray casting and enough was done for the project, the lack of implementation did not hurt the project in any way and the research will definitely prove useful for when the project is continued, thus contributing to the project's success.

Another example of the iterative approach's help was when doing the terrain generation. Due to the way Unity 5.4 handles multithreading, it was discovered quite late into the project that a most optimal performance is hardly achievable with the current implementation, but since there were other areas that the project allowed looking into, it was possible to focus on them, instead of having to stop the project and start all over in an alternative technology or trying to port the project for a later version of Unity that would target a .NET version that has its own multithreaded classes for concurrent data structure access.

The iterative approach allowed for easily getting feedback on how well the author has progressed in the project by being able to often provide complete small chunks of work being done over a period.

3.2 Technology

3.2.1 C++ vs C#

C++ is an object-oriented language that allows very good control over how memory is handled and contains many features that allow for incredible performance improvements and control over the optimisation of the code that is run. It is the go-to language when speaking of performance and video games which go hand in hand.

C# on the other hand is a higher level object-oriented language that provides a lot of the low-level implementations out of the box for the programmer. A good example is the garbage collection which is normally done automatically in C# and gives less control over the lifetime of objects, thus making C# not as good as C++ when it comes to optimisation.

One would say that the obvious choice would be to go for C++. However, referring back to Objective 5 in Chapter 1 (1.2.5 - Gain knowledge in procedural content generation for future projects), it was of a highest priority to create a tool that would be used in future projects. As Unity (which uses C#) is the editor used by the author for projects outside of university, it made less sense to create a tool in C++ which would have to be translated into C# in order to be integrated in future Unity projects and made into a game. The practicality of the created tool was the most important bit when choosing which technology to use – something that will be often mentioned throughout the paper.

3.2.2 Central Processing Unit (CPU) vs Graphics Processing Unit (GPU)

It is a well-known fact that GPUs are much faster than CPUs when it comes to rendering performance: while the average CPUs can have from 1 to 8 cores (with each capable of running 2 threads), the GPUs can have thousands of cores. The cores of CPUs are much more powerful, but the reason GPUs' slightly less powerful cores can outperform the CPU ones is parallelism. The GPU's thousands of cores can work in parallel to affect the way pixel data is displayed and work with floating point numbers very quickly. GPU power is becoming more powerful, but due to its cost, it is still not easily accessible by the average computer owner. CPUs and GPUs often work together:

> "A GPU is a purpose-built device able to assist a CPU in performing complex rendering calculations. If a scene is to look relatively realistic and predictable under virtual lighting, the rendering software should solve the rendering equation." [21]

Shaders

Shaders are small programs that are written using shading languages that are specifically designed to give instructions to the GPU on how to properly execute the code.

Because of the existence of shaders and the parallelism allowed by GPUs, it makes sense to use them when rendering or generating the large amounts of terrain data. The obvious choice for quicker procedural generation happening at runtime would be to write the execution code in shaders that would be run on the GPU. That would have worked many times better if the simulation was not a game. However, because the idea was to create a tool that can be used in a game development environment, it is important to note that even though the terrain will visually be generated very quickly on the GPU, copying the terrain data back to the CPU for generating colliders in Unity would have greatly slowed down the performance to an extent that would make that implementation questionable. Due to the practical focus of the project being probably the most important aspect, it was decided that the CPU has to be used for the terrain generation. GPU would have been an excellent option when creating a real-time visual simulation only.

3.2.3 Unity

As the author already has experience in working with Unity, it was more important to focus on the aspect of learning about the terrain generation techniques, rather than having to learn a new technology that might be able to better implement those technologies due to its underlying mechanisms (e.g. Unreal Engine with it using C++ as a development language). The learning process of the engine would have slowed down the time for exploring the topics of the paper and thus potentially leading to a lower level of knowledge gained.

Unity also does a great job at providing tools for measuring performance that fit perfectly in the study. Things like batches, draw calls, frames per second, numbers of vertices and triangles in the rendered scene were available via a click of a button which removed the need for external software. Because of Unity's Profiler and Stats windows, it was simple to record important data for this paper.

3.3 Implementation

As previously mentioned, the study involved two distinguishable sections of investigation: one being the optimal way of rendering, loading and deleting data and one being making the actual nature-like looking terrain that is optimised and visually appealing. They are separately explored in order to make it easier to understand. The tool itself is split into two to make it easier to explore aspects of the procedural generation and aspects of the different noise and their combinations.

3.3.1 Procedural Generation

Static Generation

Let us recall that a voxel "is the three-dimensional (3D) equivalent of a pixel and the tiniest distinguishable element of a 3D object."[4] With this information, the very first brute-force approach was to create a 3D array of cubes that get instantiate at the position of each voxel creating a large Rubik's style cube of cubes via the following nested for-loop:

```
public GameObject block; // variable for cube model

for (int x = 0; x < width; x++)
    for (int y = 0; y < height; y++)
        for (int z = 0; z < depth; z++)
        {
            Vector3 position = new Vector3(x,y,z);
            GameObject cube = GameObject.Instantiate(block, position, Quaternion.Identity);
        }
```

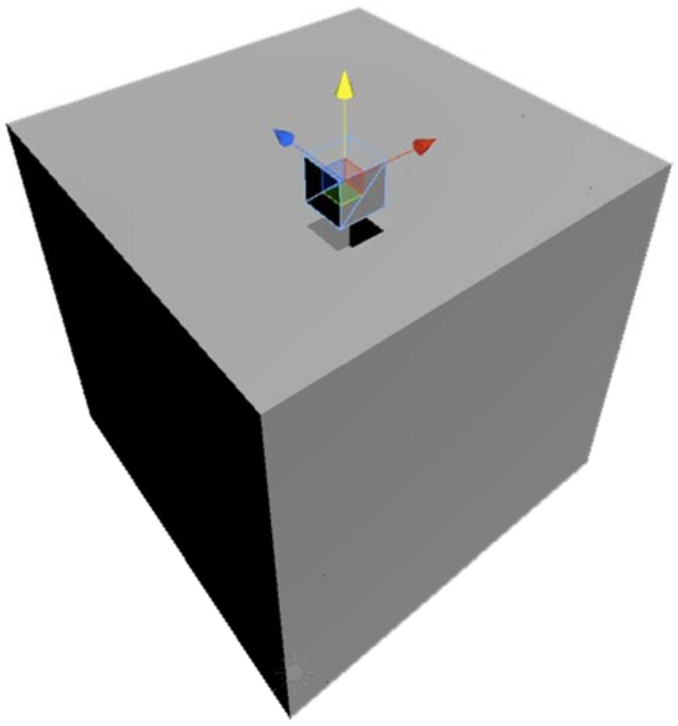


Figure 14: A cube made of a 3D array with 10x10x10 elements.

Figure 14 shows two very important outcomes if examined carefully: **Triangle count** and **Unseen polygons**

Triangle Count

Models in video games are made of triangles that are stitched together to form very complex shapes. Depending on a game's engine, graphics programmers give an estimation of available triangles that can fit in a scene. Usually, the main character model(s) are the ones that take most of the allocated triangle count with the environment taking up the rest of the available triangle count allowed per scene. Due to each game's nature and engine, the numbers of triangles allowed per scene can greatly vary. Moreover, there are things like lighting, shading, draw calls, textures and texture sizes that greatly impact the performance and polygon count is not the only thing that defines how complex a scene is to render. However, for the purpose of this paper's viewpoint, it is easy to see that the brute-force approach is very inefficient. The image above with 10x10x10 cubes contains 12 000 triangles. 10x10x10 cubes is way too small to form even a small part of a scene, let alone a massive exploration game. Scaling this cube by 10 would already make it many times bigger than some of the modern games' complex characters (Figure 15). Looking at Minecraft, with its cubes forming chunks (giant blocks of cubes combined together to form a single mesh (Figure 19)) of data being 16x16x256 blocks (16 blocks wide, 16 blocks long and 256 blocks high) [23] only a single chunk in a scene would have 65,536 blocks which is 786,432 triangles (12 triangles per cube) in a scene that is supposed to have tens or hundreds of chunks rendered at a time. It would be impossible to render such an enormous amount of data with this approach.

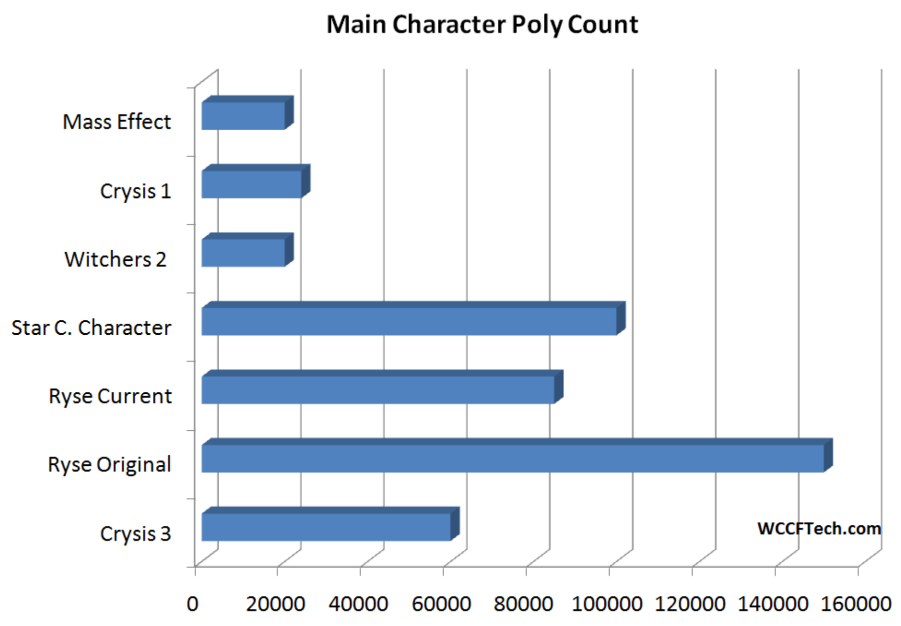


Figure 15: Comparison of polygon count of main characters in popular video games. [22]

Unseen Polygons

Figure 14 also provides a starting point for the first optimisation of the code (recall Chapter 1: objective 1.2.4: Optimise and evaluate the performance of rendering terrain). It is easy to see that many of the polygons constructing a cube will never be seen, even though being rendered in the scene. Let’s look at how a cube is made:

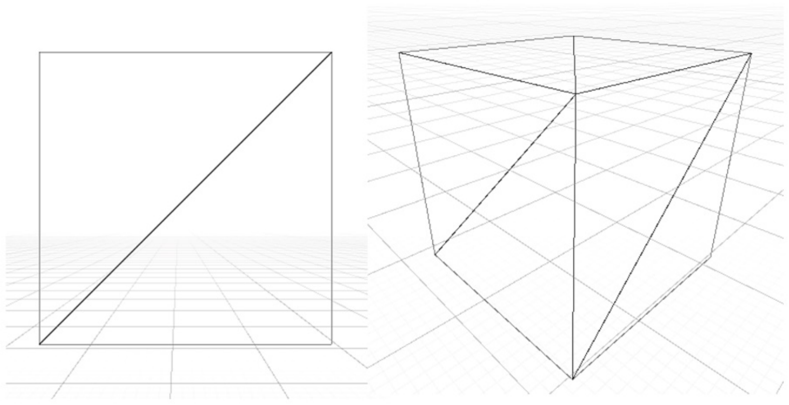


Figure 16: Left: Front view of a 3D cube. Right: Perspective view of a 3D cube.

Each cube consists of 12 triangles (2 of which make a cube’s side). On Figure 14, it is obvious that only some of the 6 sides of the cube (the top one in the extracted cube) is visible to the user as the other 5 are inside the cube. This means that the engine is needlessly rendering 12 triangles (6 sides) while it could only be rendering 2 triangles (1 side) of the shown cube. This leads to the opportunity to do the first optimisation and start removing triangles that are inside the cube.

Removing Interior Polygons

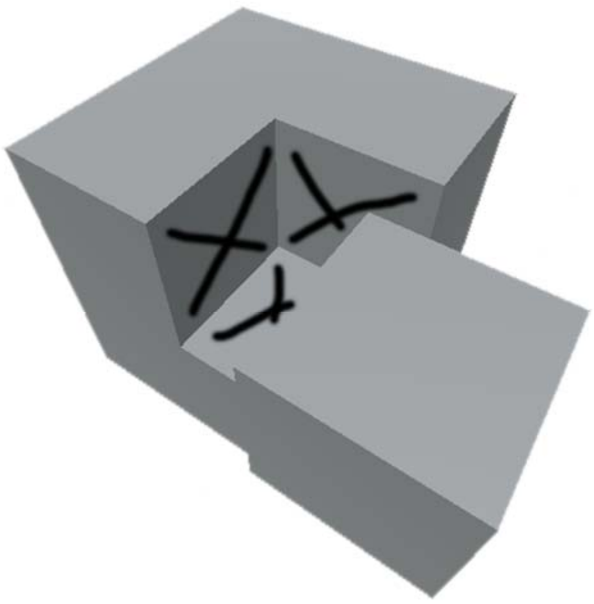


Figure 17: Removal of inner polygons to improve rendering performance.

To replace the brute-force approach which instantiates models of cubes, it is necessary to be able to reconstruct the cube. A class that creates cubes from triangles was added to the tool in order to improve performance and greatly reduce number of trianges. The purpose of this class was to make it possible for each cube to check whether it has a solid neighbour and determine which sides of itself it should render. If the cube had a neighbour – it wouldn't render the side that is next to that neighbour.

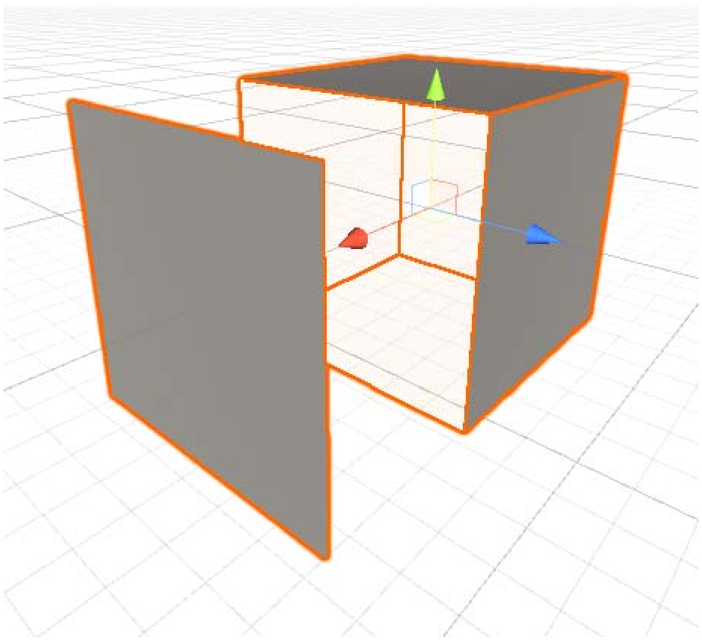


Figure 18: Manually created cube to replace cube model instantiation.

In order to further optimise the implementation, once the cube was manually created and the big cube of cubes created, the meshes of all the cubes were combined into a single big mesh called a chunk. This introduced one of Unity's restrictions that a mesh cannot have more than ~65000 vertices. Since a chunk in Minecraft is made of 65536 cubes, technically Unity can never make a single chunk with that size. However, by recombining the meshes multiple chunks could be stack on top of each other to achieve a similar effect.

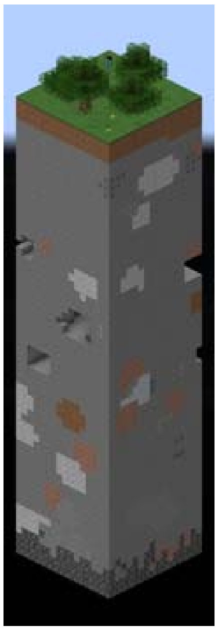


Figure 19: A 16x16x256 blocks Chunk in Minecraft. [24]

Because the world had to have tens or hundreds chunks at the same time, initially the chunks suffered from the same issue as the cubes – neighbouring chunks did not know about each other so the code was only placing them next to one another without any optimisation. A further improvement was made with making the chunks aware of their neighbours and having them remove the inner polygons that were never to be seen. (Figure 20)

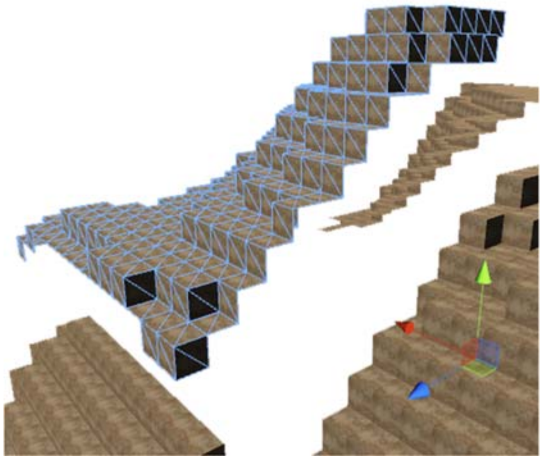


Figure 20: A screenshot of the chunk optimisation done inside the developed tool.

The way the optimisation for both chunks and cubes' neighbours was achieved by constructing the world cube by cube and passing to it its neighbour's coordinates (if one existed) and doing a check against each of the cube's sides as it was created. This implementation was useful later on during the Marching Cubes implementation too as the algorithm works on cubes one at a time (hence the "marching" name).

The only thing left was to see how large of a static world could be built using the optimisations. The results will be presented in the next chapter.

Dynamic Generation

Moving onto the dynamic generation was a result of building up on the knowledge of the static generation gained so far.

The first step towards a never-ending world was adding a simple radius value to the player.

```
for (int x = -radius; x < radius; x++)
    for (int z = -radius; z < radius; z++)
        for (int y = 0; y < worldHeight; y++)
        {
            // Code to add the chunks at the position
        }
```

A simple code change that generates the world around the player to achieve the effect shown on Figure 5.

The radius value was getting updated as the player moved which made it possible to use the player's position and the radius around them to determine a label for each chunk in the world around the player. Existing chunks were kept in a dictionary data structure, due to its fast access speed and the ability to add or remove elements quickly. Each chunk could have one of three labels: KEEP, DRAW and DONE. Every time a player moved, each of the possible chunk positions around them would be checked against the dictionary of chunks. If a chunk existed in the dictionary and was within the radius, it would be kept for the next frame; if it didn't exist, it was getting created; if it existed, but was not within the radius – it would be marked as DONE and get destroyed.

This sounded like a well-oiled solution and seemed as if the whole generation issue was solved. However, this solution was extremely slow and inefficient. It could have been sufficient enough if it involved a non-open world with different levels that allowed some time for new terrain to be loaded (e.g. the dungeons in Diablo). However, this wasn't the case for the open world game and an alternative solution had to be implemented. This part is where the GPU would have done a much better job but would have been limited to being a simple simulation without any collision detection to be done on the CPU (as previously – copying data from the GPU to the CPU is costly, and in order to generate colliders, the CPU must know where exactly was the data generated by the GPU).

Recursive Flood Fill
The problem lead to the need for multithreaded generation on the CPU. But what is multithreading?

"In computer architecture, multithreading is the ability of a central processing unit (CPU) (or a single core in a multi-core processor) to provide multiple threads of execution concurrently, supported by the operating system." [25]

This meant that simultaneous chunks could be built around the player and it was possible to significantly reduce the loading and deletion time of terrain within a radius of the player. Recall from Figure 7 that what the Flood Fill algorithm does is present a top down grid of the terrain with the player being in the middle. Then a search in 4 directions gets initiated and each location recursively calls the implemented algorithm until the criteria for termination has been reached. The criteria for this particular implementation was reaching the limit given by a radius around the player. The multithreaded implementation involved the use of a concurrent dictionary and a concurrent queue which allowed for multiple threads to access chunks simultaneously without causing deadlocks.

This implementation made it possible for data to be dynamically loaded as the player was moving and the information was loading at a very good pace. When tested on a relatively new machine, this code was executing perfectly. However, when tested on an older CPU with less cores/threads, it was sometimes possible for the player to run out of the world and fall. One possible solution for the issue was slowing down the player which gave more time to the CPU to generate new chunks. Because this tool was developed in Unity 5.4 which only targets .NET 4.6, custom concurrency queue and dictionaries had to be implemented (borrowed from Nicholas Ventimiglia [26]). Newer versions of Unity target .NET 4.7.2 which has its own versions of ConcurrentQueue and ConcurrentDictionary [27] and it is possible that these outperform the current implementation. This issue is further discussed in the section for future work towards the conclusion of the paper.

Combining Static and Dynamic
A combination of the two techniques was achieved by doing an initial world building when launching the application which takes a certain time depending on how big the initial world is desired to be. As the player approaches the edges of this world, the dynamic generation will kick in depending on a radius value around the player which will generate new chunks of the terrain and delete the old ones.

3.3.2 Nature-like terrain
Once the voxel data generation, storage, loading and deleting was implemented and the techniques compared, it was time to start making terrain-like surface and not only giant cubes of cubes. When it comes to using noise in data generation, Perlin noise is the go-to option with its countless implementations and available tutorials online.

Perlin Noise
A brief introduction was given during the Background and Research chapter on the different types of noise that were studied in this paper. The most important thing to remember was that noise is pseudo-random (for any known input, the result that

comes out will be the same). Because of this property, it is very easy to implement the noise into terrain generation – knowing that the same output will be produced by giving a certain seed (integer number) means that terrain data can be preserved during frames and will not change. It is very straightforward to come up with visually appealing results by simply trying out different seed values and selecting the most interesting ones. It is important to note here that the visual appearance is, of course, biased and based on the perception of the person examining the results.

Two other properties of the noise are amplitude and frequency. Thanks to amplitude (the difference between the wave's peaks and drops) and frequency (how often these peaks and drops repeat) the noise's outcome could be controlled during implementation. These two properties were altered to achieve different types of terrain in the tool: increasing the frequency, for example, will create very steep terrain whereas lowering the amplitude will create a very flat terrain. The waves' amplitude and frequency properties are particularly useful when implementing the Fractal Brownian Motion which is examined further in the chapter.

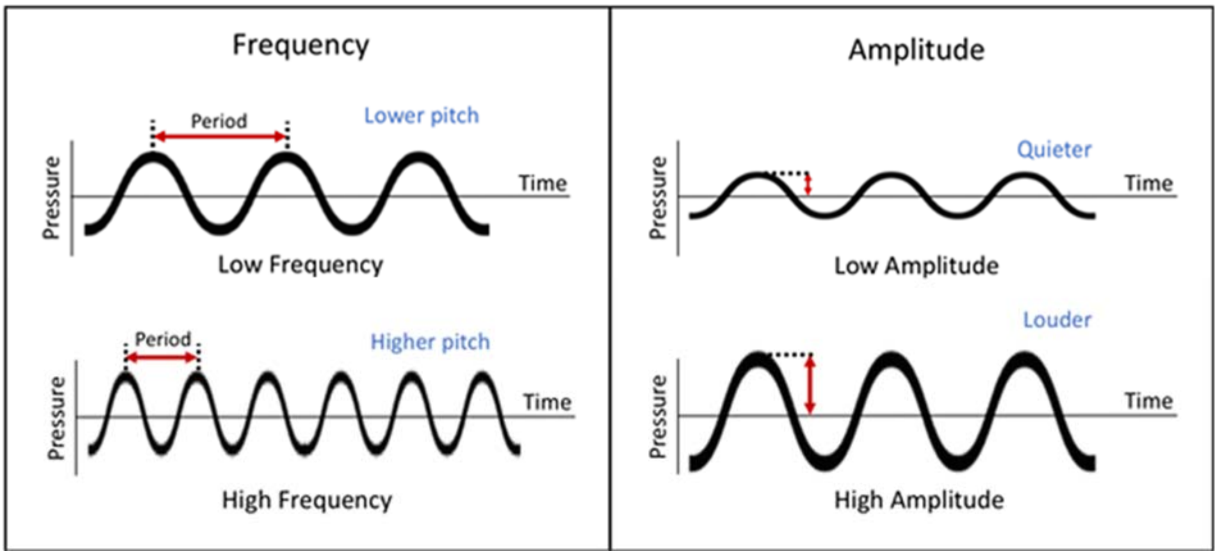


Figure 21: Waves with varying amplitude and frequency properties. [28]

Another property of the Perlin noise is that it can be tiled/periodic. This means that it can repeat itself in order to create large amounts of data. It can be seen as a wave that can be zoomed in or zoomed out and sampled. Zooming in would mean a repetition that happens too often making the terrain look too repetitive. Zooming out too much can mean that a lot of details get missed out however because the repetition will be made hard to spot. A good middle point has to be found when using the different variables of the noise. All these properties of the noise were important to understand because they are part of the tool and can be tweaked to achieve different variations of the terrain. In order to understand the results presented in the next chapter, these properties of the noise should be.

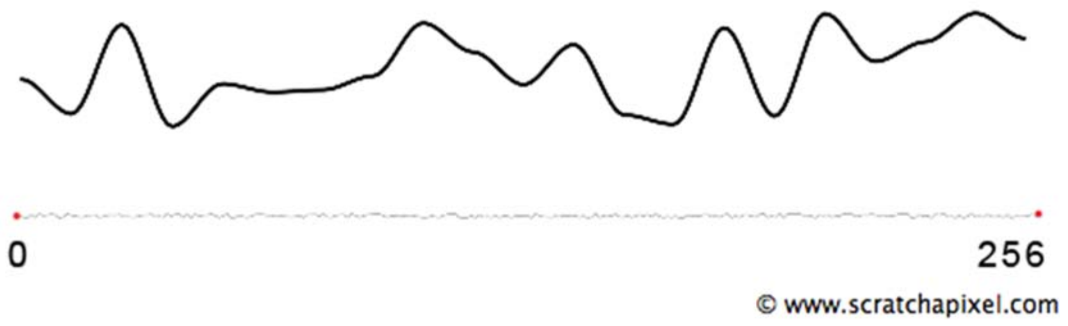


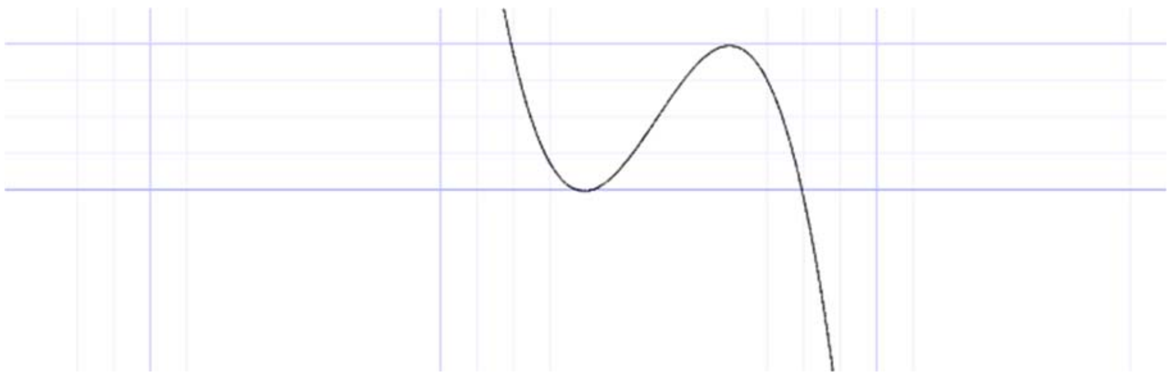
Figure 22: A good noise length of 256 making repetition hard to spot [15]

If too small of a value is chosen (e.g. 20 random numbers), the numbers will repeat way too often (in a 300 units period, the numbers will repeat 15 times) and make it too obvious that there is a repetition pattern. After some experimentation, Ken Perlin decided to go for a good number of length with 256.

In order to come up with a data that represents the random numbers that are initially generated when starting off the Perlin noise implementation, Ken Perlin came up with a permutation table that contains a total of 512 numbers. In order to achieve a good tiling period of the noise, Ken Perlin decided to put in the numbers from 0 to 255 in a random order in which each number is contained only once. Then he repeated these numbers to fill in the rest of the values of the permutation table which made the table long enough to allow for the noise to be repeated without being recognised and thus forming good non-repetitive patterns. This permutation table has been used in the implementation of this paper.

Improved Noise

This noise replaces the original smoothing function in Perlin noise from f(x) = 3x^2-2x^3 to f(x) = 6x^5-15x^4+10x^3 in order to improve the curve of the function and achieve better visual results.



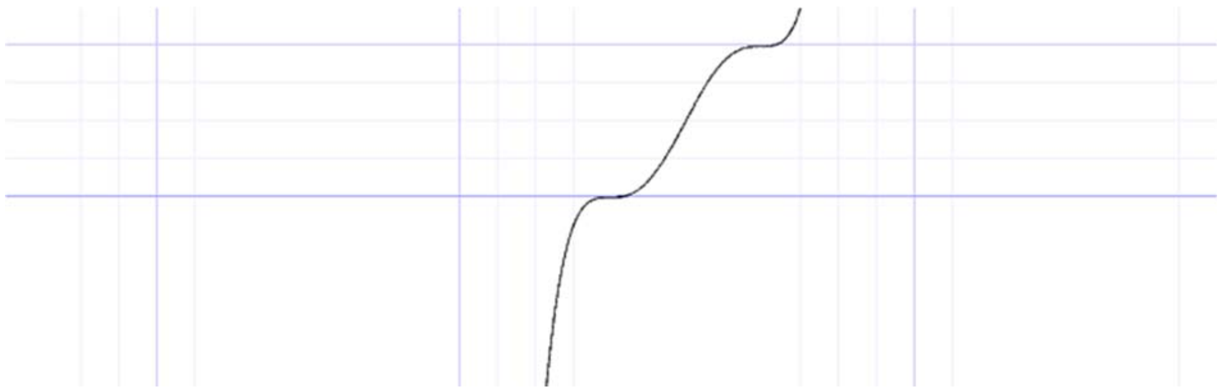


Figure 23: Top: Old smoothing formula. Bottom: New smoothing formula. [16]

This is the version of the Perlin noise that was used for the tool's implementation due to its proven better visual results and performance.

Simplex Noise

Let's recall the advantages of Simplex noise over the classic Perlin noise:

"- Simplex noise has a lower computational complexity and requires fewer multiplications.

- Simplex noise scales to higher dimensions (4D, 5D) with much less computational cost: the complexity is O(n^2) for n dimensions instead of the O(n2^n) of classic noise.
- Simplex noise is easy to implement in hardware." [19]

Due to its faster computation, the Simplex noise was used to compare against the classic Perlin noise. Even though the noise cannot be used in an implementation for commercial use as it is patented, it served a great purpose for learning an alternative implementation of the Perlin noise and learn about the different aspects of noise and the various places where it can be improved.

Fractal Brownian Motion

While not an actual noise, it does use combinations of same or different types of noise with varying values to achieve realistic and outstanding results. Each of these layers contain variations of amplitude and frequency. In order to work with Fractal Brownian Motion, the concept of octaves must be introduced. By definition, an octave of noise is a wave which has twice the frequency of the previous wave. However, the rule of doubling the frequency is not always kept and the term of octaves is misused due to the variations still achieving great-looking results.

"By adding different iterations of noise (octaves), where we successively increment the frequencies in regular steps (lacunarity) and decrease the amplitude (gain) of the noise we can obtain a finer granularity in the noise and get more fine detail. This technique is called Fractal Brownian Motion" [29]

The FBM with its variables for number of octaves, lacunarity and gain were added to the tool in order to be able to explore the great results that can be achieved by this technique.

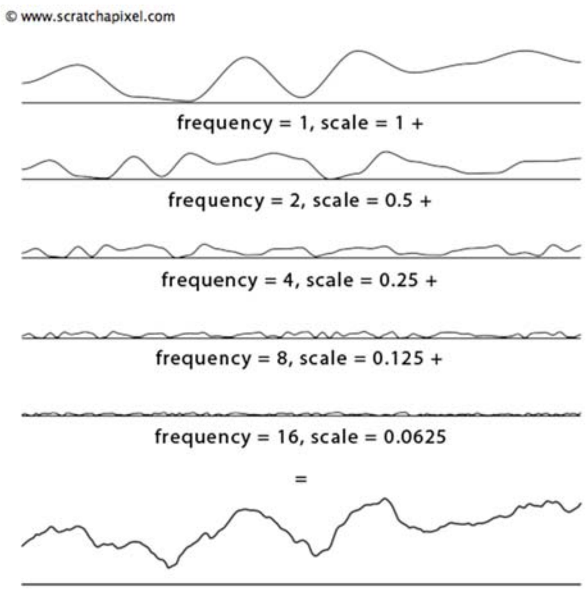


Figure 24: A combination of 5 noise layers to achieve Fractal Brownian Motion wave that represents a mountainous terrain.

Even though FBM is more computationally intensive because it has to sample many noise waves, its results are much more realistic and stunning. As Figure 24 shows, the combination of 5 layers produces a wave that closely resembles natural terrain. Once generated, this wave can be sampled at various intervals and the values gained from sampling used to determine the Y-value (up direction) of the terrain.

Marching Cubes

"Marching cubes uses a divide-and-conquer approach to locate the surface in a logical cube created from eight pixels; four each from two adjacent slices." [20]

Isosurface is a 3D version of a contour line which represents a certain elevation on a map. This elevation can represent the surface of the terrain data. In the dissertation tool's implementation, the isosurface value is generated by the noise functions depending on where they are sampled.

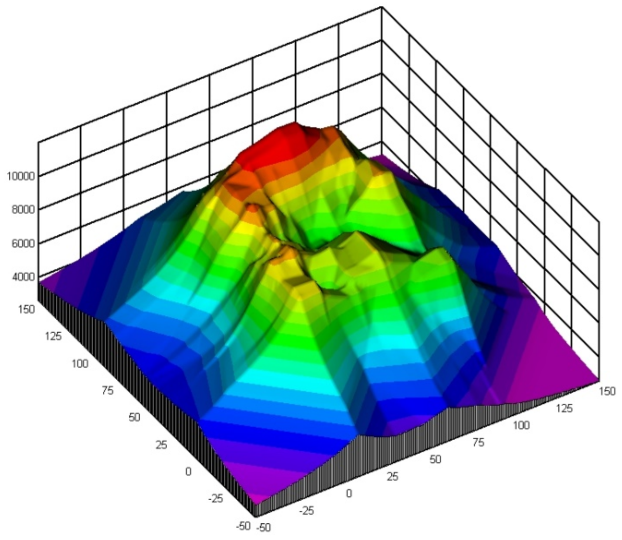


Figure 25: Contour lines whose data can be used to extract as terrain height data and used as isoline value in the Marching Cubes algorithm. [31]

What the algorithm does is walk through a field of voxels (one at a time), creating imaginary cubes with 8 neighbour vertices at a time. Each of the 8 vertices' values are checked against the isosurface value and saved to an 8-bit index containing a bit for each of the vertices. If a vertex's value is above the isosurface – it gets saved as 0, if below – it is saved as 1. Then this 8-bit index is passed to a piece of code that looks up a special table called edge table and retrieves exactly one possible combination for the given index. The table returns between 1 and 4 triangles for that corresponding index, some combinations of which are shown in Figure 26. There are a total of 256 combinations, many of which are simple symmetries of the initial 15 cases.

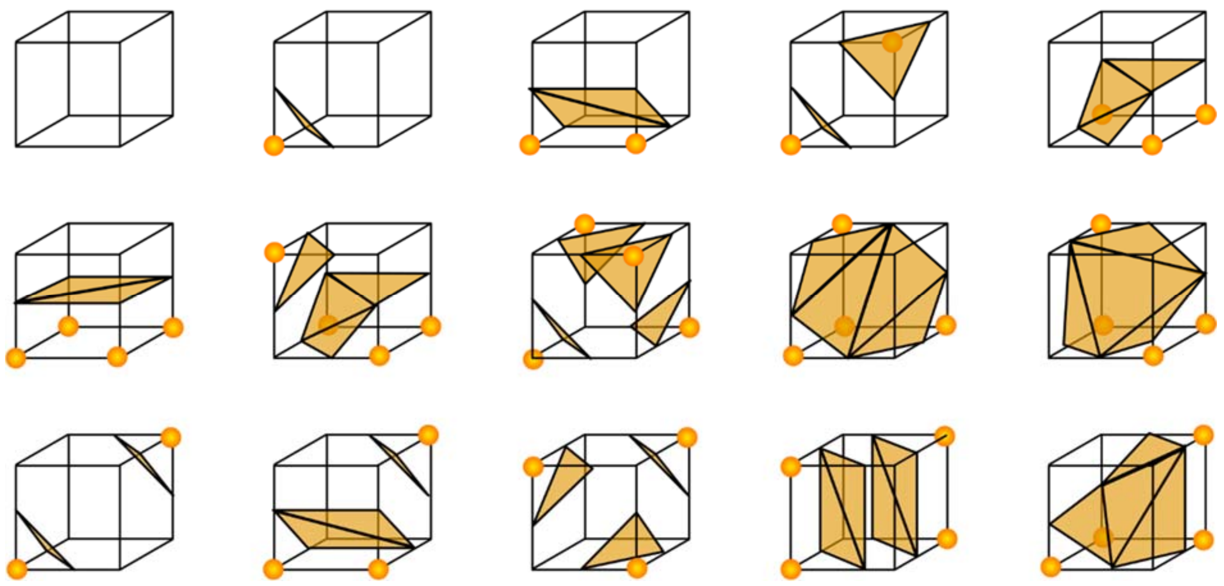


Figure 26: The originally published 15 cube configurations. [30]

After the index is used to tell what are the edges of the surface, linear interpolation is used to determine how exactly the surface should look. The given edges are used to lookup data from another table that returns a specific triangulation from the combined edges.

The algorithm finds its use in the research paper by being implemented after the noise functions form the general shape of the terrain. Once the shape is created, the Marching Cubes make it go from the blocky terrain into a very smooth version that can be used to create highly complex terrain data. Some examples will be shown where the Fractal Brownian Motion and Marching Cubes algorithms are combined to form very sophisticated cave models.

Volume Ray Casting

Volume ray casting or ray marching (not to be confused with raytracing which is too expensive for real-time simulations that games are) is an alternative technique for rendering the terrain generated by the tool. As opposed to marching on cubes, this technique marches along a ray that is cast from the camera's position until it reaches a point where it intersects with an object.

"It turns out that sampling this point along the ray is a relatively simple and inexpensive operation, and much more practical in realtime… this method is less accurate than raytracing (if you look closely the intersection point is slightly off). For games however it is more than adequate, and is a great compromise between the efficiency of polygonal rendering and the accuracy of traditional raytracing." [32]

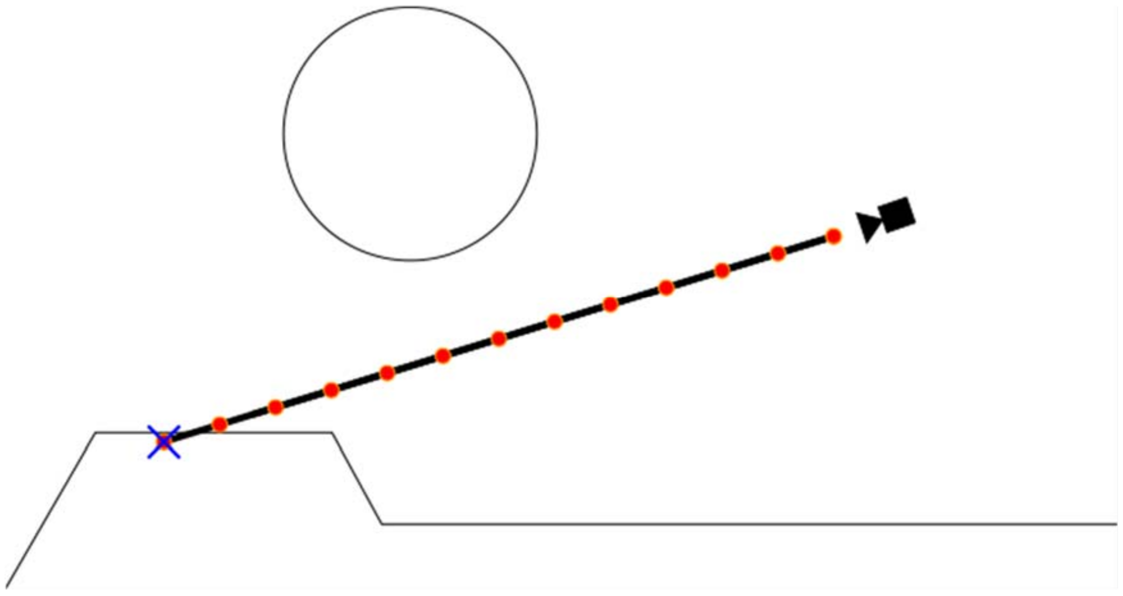


Figure 27: Visual representation of raymarching. [32]

There are 4 steps in the algorithm's work: [33]

1. Ray casting which casts the rays from the camera and continues iterating until a shape is hit. (as shown on Figure 26)
2. Sampling is done across the cast ray (red dots on Figure 26). Because the red dots might be off the surface of the shape, linear interpolation is used again to find out where exactly the shape's surface is.
3. Once the surface is hit, its RGBA values must be retrieved and the surface should be shaded appropriately depending on where the light source in the scene is.
4. The proper composition from the various sampled points is created to show the resulting pixels with their appropriate colour and shading values.

The raymarching technique runs on every pixel of what the camera sees, therefore it should be implemented on the GPU in a post-processing shader. Because of the necessity to use a shader, this technique requires a GPU-based implementation. The technique was only investigated in theory due to the lack of time remaining for the project's implementation (another benefit of the iterative approach where the success of the project was achieved even if not enough time was left to implement this technique). If an approach to use GPU rendering is chosen for future use of the tool, this technique will definitely be of a great use.

3.4 Summary

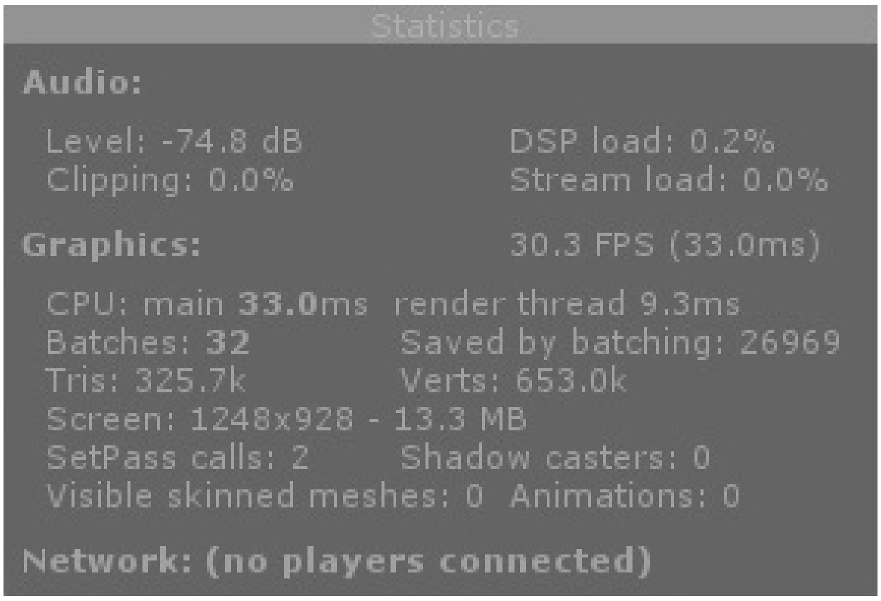
This chapter was about the development, decision making and implementation of the tool and choosing Unity that can be used to generate terrain for a video game. The iterative development strategy and methodology was presented and compared to waterfall methodology. What followed was a discussion of the available technologies like programming languages as well as comparing the CPU and GPU approaches. Unity was chosen as the main working environment of this paper due to its future practical purpose. Some arguments were discussed in favour of the different available technologies and methodologies and the reasoning behind choosing some over the other were presented. The implementation section tried to give a high-level brief overview of each of the implemented technologies and best effort was made to omit unnecessary details unrelated to the implementation. A timeline-like walkthrough of the tool's implementation and the logic behind the optimisation changes implemented was presented along with helpful figures and code snippets where possible to get each point across.

Chapter 4: Results and Evaluation

This chapter talks about the ways the different techniques were compared against each other and evaluated. A brief explanation of the recorder metrics is presented to give an idea of how the tool compares the different approaches. Then follow the results that have been achieved based on two scenarios investigated with the help of the tool. Some figures showing successful renders of terrain done with the tool are shown, followed by a section that validates how each objective of the paper was met.

4.1 Recorded metrics

In order to ensure a proper evaluation was done, there needed to be some common and constant values shared between the techniques that were tested. All the comparisons were done on the same machine and when comparing the different techniques, the terrain size variations were the same (e.g. when comparing noise functions, the size of the terrain grew in increments that were the same for each tested noise).



```
Statistics
Audio:
  Level: -74.8 dB                 DSP load: 0.2%
  Clipping: 0.0%                  Stream load: 0.0%
Graphics:                         30.3 FPS (33.0ms)
  CPU: main 33.0ms  render thread 9.3ms
  Batches: 32             Saved by batching: 26969
  Tris: 325.7k            Verts: 653.0k
  Screen: 1248x928 - 13.3 MB
  SetPass calls: 2        Shadow casters: 0
  Visible skinned meshes: 0  Animations: 0
Network: (no players connected)
```

Figure 28: Unity's stats window used for measurement.

The recorded metrics when doing the evaluation were:

Frame rate

Frame rate (measured in frames per second) is the frequency at which frames appear on the displaying device. Frames per second (FPS) measurements give feedback at how many frames can be shown per second thus giving a good indication at how fast calculation is done. The higher the frames, the smoother the picture. Therefore, in real-time simulations like video games, higher frame rates are important. Generally rates around or more than 60 FPS are considered a good performance, although 30 FPS is not rare to be seen in some video games, especially on consoles. It was important to record the average FPS and not the highest achieved one because a fluctuating frame rate between 30 and 120 is not considered well-optimised and does lead to inconsistency when playing. In order for terrain generation technique to be considered successful, the average FPS had to be at least above 30 and ideally above 60.

Time per Frame

Time per Frame measures the time it takes for a frame to load. The lesser it takes, the more frames can be fit into a second of game time. It is often suggested to use frame time over frame rate [34] but both have been presented in the evaluation due to FPS being much more widely used and easier to understand by simply knowing that 60+ FPS is a very good outcome and 30+ is acceptable but not very optimal. Times per frame can be difficult to understand (is 5ms too quick or too slow to load) but still useful to show the increase or decrease when optimising. In Unity, time per frame “only includes the time taken to do the frame update and render the game view; it does not include the time taken in the editor to draw the scene view, inspector and other editor-only processing.”[35]

Vertices and Triangles

It was mentioned during the implementation chapter that the amount of triangles and vertices per frame are an important aspect of a game’s performance and put a limit to the artists’ creativity. Performance versus aesthetics is a very common problem to solve in video game development and due to the limits of triangles that can be sometimes put by developers, it was important to make sure that this metric is recorded. Due to the way noise is implemented, it was important to observe not only the visual result and frame rate but the structure of a mesh. This metric was important when exploring the different combinations of noise with the mesh extraction after Marching Cubes algorithm is run. If the evaluation was simply done based on the frames per second, the results of the different noise techniques would not have varied a lot but the vertex and triangle count of the mesh did vary greatly, which made it possible to show how big a scene can become before the tool drastically dropping the frames per second.

Batches

When the CPU needs to communicate to the GPU and share data with it (so that the GPU can render the screen content) it can send things in batches. For example, if many objects share a single texture, the CPU can optimise this group of objects and send their data together to the GPU and thus save a lot of changes by “batching” the data together. This measure was initially taken when comparing the performance of the terrain generation methods, but since the project focuses more on the terrain generation techniques like generation, noise and extracting meshes, not a lot of attention was paid on the batching because it takes into consideration aspects of the game like lighting and texturing which often are optimised by batching but were not part of this paper’s investigation.

4.2 Procedural Generation Evaluation

4.2.1 Static Evaluation

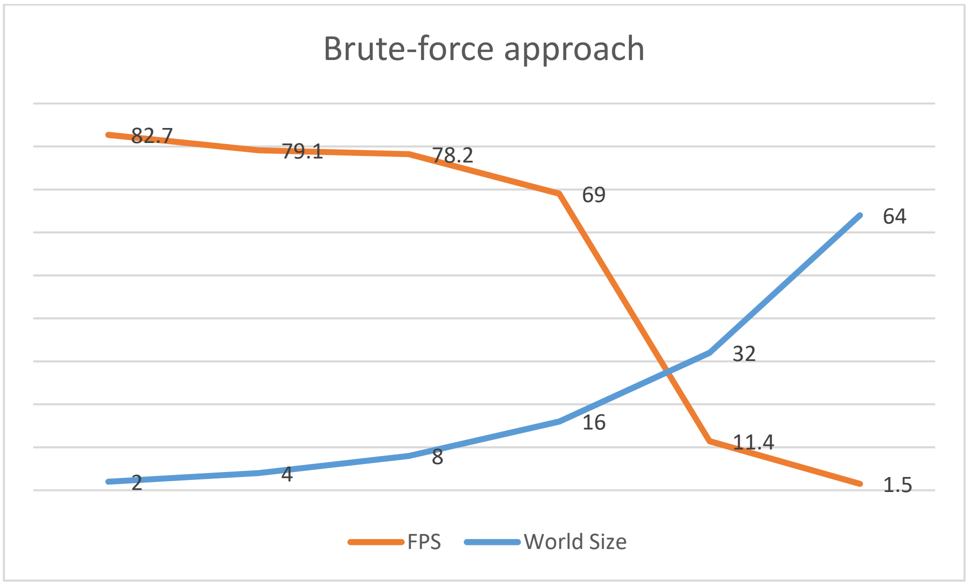


Table 1: Performance measured on the brute-force approach.

The brute force approach of instantiating cubes without any optimisation was quickly proven inefficient. As the world size went to 16x16x16 cubes it was keeping frame rate around 69.0 but doubling the world in size made the FPS dive and make it unbearable to navigate through. Making the world 128x128x128 cubes crashed Unity.

Removing Unseen Triangles

After the two optimisations for removing the inner sides of cubes and chunks that are neighbouring, the world that could be created was drastically made bigger.

World Size	FPS	Frame time (in ms)	Triangles
2	87.7	11.3	280
4	87.2	12.4	1900
8	86.7	11.2	2400
32	79.7	12.6	5400
64	2.4	418.1	50800

Table 2: Performance measured on the optimised blocks and chunks approach.

The optimisations proved to have a better overall performance by maintaining better frames per second and frame time as well as greatly reduce the triangle count of the created terrain data. However, going into terrain with size 64x64x64 managed to greatly slow down Unity and going into an even larger number (128x128x128) managed to crash Unity.

A world of less than 64x64x64 cubes is nowhere near big enough to hold any modern commercial video game's terrain data which made it impossible to continue with the static world generation approach. The investigation of dynamic generation began.

4.2.2 Dynamic Evaluation

When the dynamic generation was initially implemented, it was running on a single thread which meant certain periods of time had to pass before new data gets generated [36]. This can be seen on the video shown in the reference which was uploaded by the author to show that it could sometimes take up to 10 seconds to generate the new data. In practice, sometimes it would take even more than 15 seconds. This could work with a game that is split in different levels, with enough time provided for a level to load, but not in an infinite world game. This lead to the need for multithreaded implementation. When the Flood Fill algorithm was implemented, the terrain was loading much faster on the tested machine.

However, this was also tested against an older machine and it proved to be inefficient and still not fast enough. Using the Flood Fill algorithm allowed for very brief moments of delay on the old machine – sometimes up to 2 seconds, which was still much better than having to wait tens of seconds for a chunk of blocks to load.

As briefly mentioned before, due to the version of Unity (5.4) being targeted at .NET 4.6 and not .NET 4.7 (which provides concurrent access to dictionaries and queues) the multithreading had to be implemented manually which could potentially be less optimised and performing worse than the classes provided with .NET. Due to the tool working well enough on the tested machine, the time restriction and the bigger emphasis on the data generation with noise, it was decided that the generation is good enough to allow moving forward towards the comparison of noise functions. Some improvements are suggested in the future work part of the Conclusion chapter.

Noise Evaluation

The noise generation was compared in two scenarios. In both scenario there were two equally important things to compare – performance and visual impact. As mentioned before the quality of art versus performance is a common debate in video games development and it was expected that even if one technique outperforms another, the slower one should not be immediately thrown away as it might be producing better visual results.

4.3 Evaluation Scenarios

4.3.1 Scenario One

In scenario one, a comparison between Perlin and Simplex noise in combination with the Marching cubes algorithm was done. In addition, variations of the noise techniques with Fractal Brownian Motion was compared to the other techniques. The scene used for comparison of the performance and looks was one that resembled a cave system of some sort, instead of simply showing planes of terrain data:

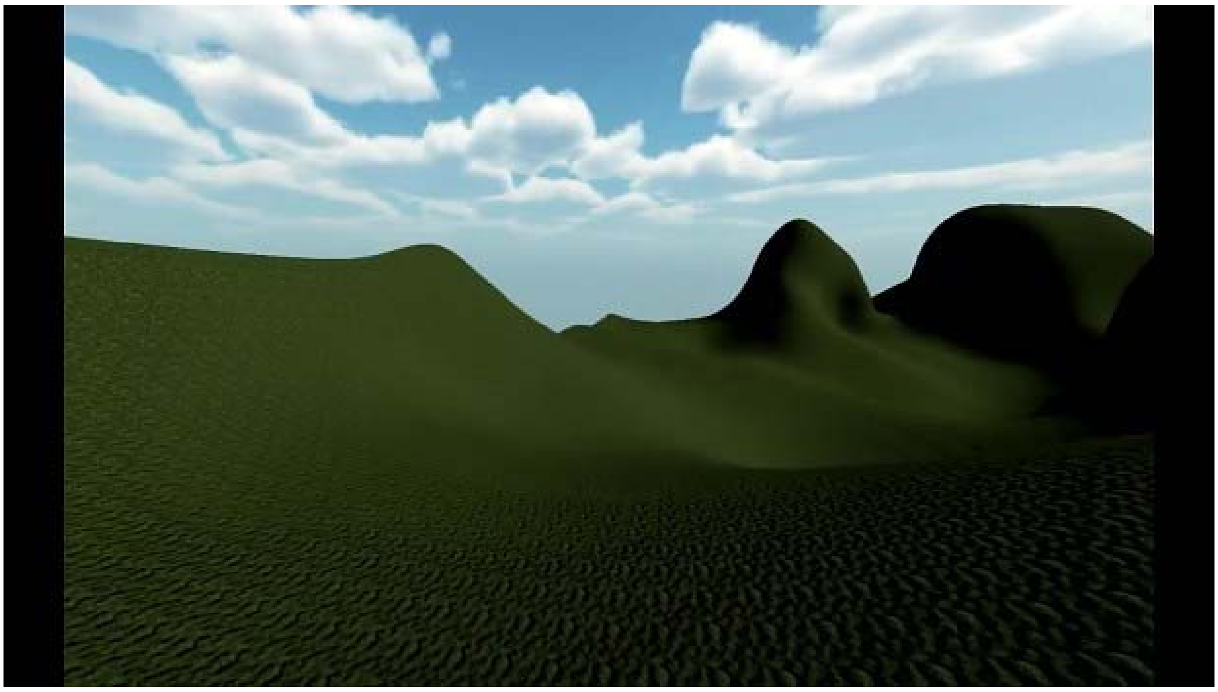


Figure 29: A plane with extracted terrain mesh data using Marching Cubes. [37]

The plain old planes were used in Scenario Two but looking at simple planes makes it hard to compare the visual appeal of generated data so a cave-like shape was chosen to hopefully show some complexity and variation in the generated data with the different functions.

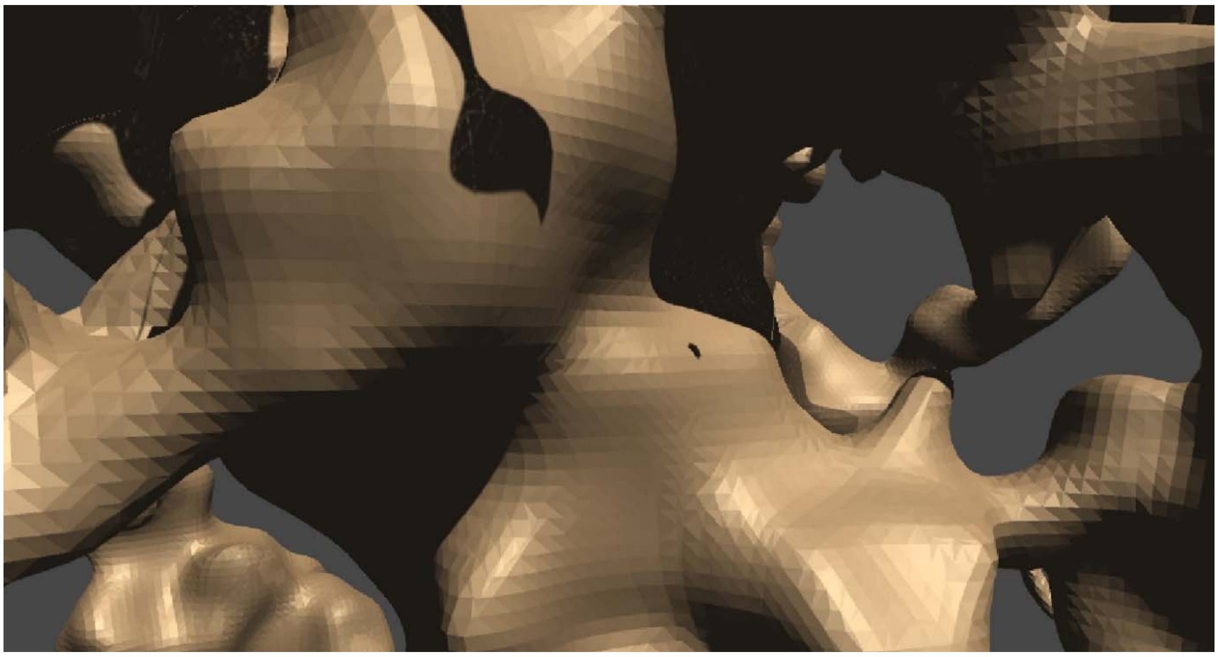


Figure 30: A cave-like system produced by the tool using Fractal Brownian Motion combining 2 layers of Perlin noise.

When the noise functions were tested for their performance, a seed value of 1 and a frequency of 3.0f was used. For the Fractal Brownian Noise, 4 octaves were chosen with the respective Perlin or Simplex noise passed. Results were taken from a machine with an Intel i7-4771 @3.50GHz CPU with 4 Cores and 8 Logical Processors and 16 GB of DDR3 RAM.

Pure Noise Functions Evaluation

Perlin noise					Simplex noise			
World Size	FPS	Frame time (in ms)	Tris		World Size	FPS	Frame time (in ms)	Tris
16	81.1	12.3	17 100		16	87.2	11.5	12 700
32	81.4	12	58 700		32	84.3	11.9	53 100
64	80.6	12.1	221 000		64	82.1	12.2	178 100
128	78.9	12.2	657 200		128	77.9	12.8	603 300
256	78.3	12.8	2 000 000		256	79.3	12.9	1 800 000
512	44.4	22.5	19 000 000		512	59.6	17.3	13 900 000

Table 3: The results of comparing performance of Perlin and Simplex noise functions.

Table 3 shows proof that the Simplex noise massively outperforms Perlin noise when it comes to the number of triangles on the scene. Simplex noise can be very cost-efficient especially as the world gets bigger – 13.9 MM (million) to 19 MM is a very good optimisation. In addition, it can be seen that the FPS is better although it cannot be easily noticed until the world becomes quite large where the difference is significant with 15.5 frame gain in a 512x512x512 world with Simplex noise. Simplex noise scene was also faster to load initially when starting with 3D world of size 256 or 512 but it is, again, largely affected by the number of triangles being less considering the triangles being millions at that point.

Noise Functions with Fractal Brownian Motion Evaluation

Perlin noise with FBM					Simplex noise with FBM			
World Size	FPS	Frame time (in ms)	Tris		World Size	FPS	Frame time (in ms)	Tris
16	82.1	12.2	40 100		16	77.9	12.2	38 200
32	77.7	12.9	169 400		32	77.2	12.8	149 300
64	76.3	13.1	587 200		64	75.9	13.2	537 900
128	72.5	13.7	1 900 000		128	74.9	13.4	1 500 000
256	59.5	16.8	6 300 000		256	69.8	14.3	5 400 000
512	N/A	N/A	~40 000 000		512	N/A	N/A	~30 000 000

Table 4: The results of comparing performance of Perlin and Simplex noise functions with Fractal Brownian Motion.

From the table it can be clearly seen again that Simplex noise was the winner in the evaluation of the noise functions with FBM when it comes to the triangle count, although with not such a big difference. The triangle count reduction was not in a very big percentage. Moreover, the FPS gain from using Simplex noise was not extremely convincing, although still better overall.

At the end of the day, Simplex noise was the clear winner in scenario one, but when it comes using the noise layers with FBM, one could argue whether it is worth going the path of Simplex noise for the small gain it provided. It is important to recall that Simplex noise is considered giving even better results in higher dimensions, so theoretically it is much better when it comes to 4D + dimensions. However, such an evaluation is not considered in this paper.

4.3.2 Scenario Two

In scenario two, the practical focus was towards the outline of the generated terrain and its use in a development environment. A reminder that the most important outcome for this tool would be its ability to be used as part of development of future games. This made Simplex noise a less desired outcome due to the author finding out it is patented which practically makes it problematic to use. Since Perlin noise was proven in Scenario one to not be much worse than Simplex noise when working in 3D sampling environment, it was chosen to use Perlin noise for various combinations of layers that Fractal Brownian Motion can use to create unique terrains.. Obviously due to FBM having to sample multiple layers of noise it was slower than just using Perlin noise (as was proved in Scenario One) but this scenario's aim was to explore the chances of ending up with much better visually appealing results while being aware of the performance hit that the generation will have. In simple words – this was just a pure evaluation of visual appeal of the results. It is very important to note that visual aesthetics are a subjective manner and what the author has found appealing does not necessarily reflect the truth.

For the results to be understood, a few terms need to be explained:

Octaves as mentioned before are the number of layers of noise to sample. For every new octave, the **frequency** doubles (**lacunarity** factor) and **amplitude** halves (**gain** factor). There is a **smooth** factor that determines how often the data is sampled. For the results shown below, the smoothness factor is of 0.1 value. If the value is too small, then only lower parts of the noise will be sampled, and if it is too big – only higher parts will be sampled. It can be imagined like zooming in and out of the graph respectively. 0.1 is a good medium value that gives nice results. **Persistence** controls how much of an impact each next sampled layer has.

The more octaves that are added together, the more functions are sampled which leads to more jagged edges and more realistic world. Depending on the game art requirement, a different effect can be achieved.

Below are shown screenshots of evaluating a 16x16x256 terrain block which implements different number of octaves. The left images will be showing a perspective view of terrain generated by the tool and the right images will be showing the silhouette of this terrain to make it easier for the reader to spot the wave's shape. All the measurements are taken with a persistence value of 0.5. The last image shows the large impact of the persistence value when sampling the noise.

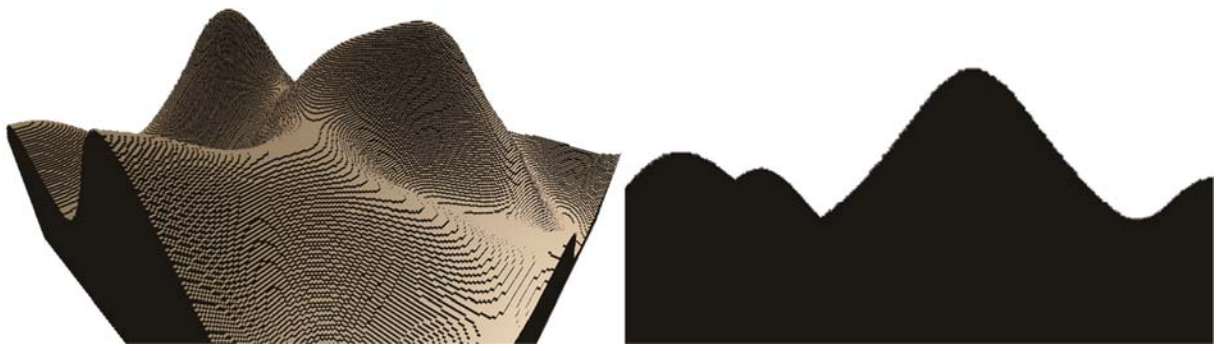


Figure 31: Generated terrain with 1 octave.

Simply put – the outcome of using Perlin noise only. As it can be observed, the terrain data looks very smooth without any jagged edges.

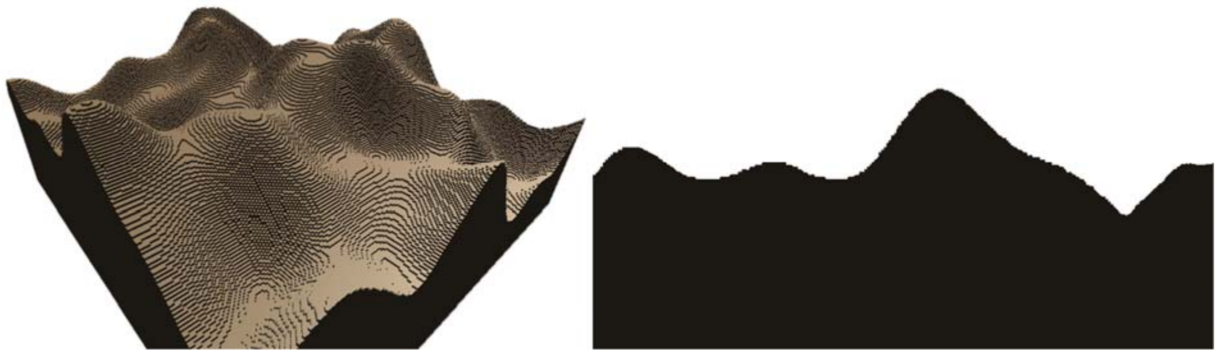


Figure 32: Generated terrain with 2 octaves.

Adding 2 octaves already shows how much more detail gets added by simply looking at the amount of hills that get added. The terrain starts looking very natural.

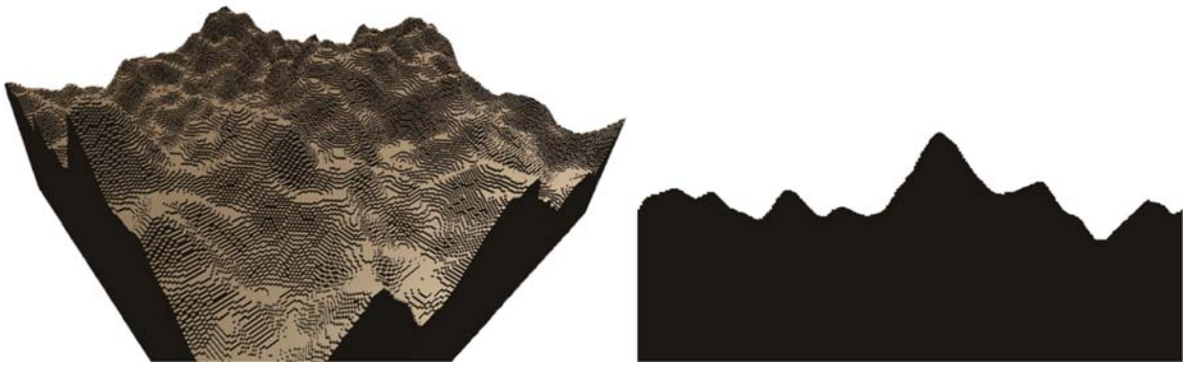


Figure 33: Generated terrain with 4 octaves.

Increasing the octaves to 4 adds great detail to the terrain, increases jaggedness of the hills and adds great variation of heights.

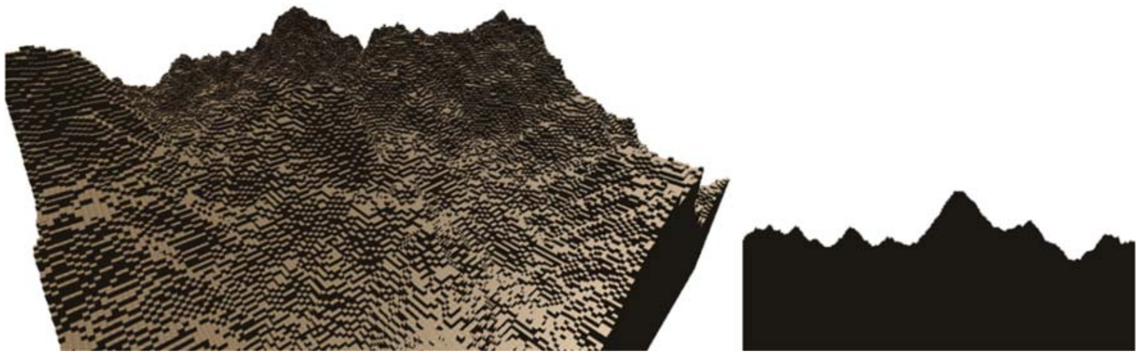


Figure 34: Generated terrain with 8 octaves.

Using 8 octaves brings the details even a step further, even though it starts adding a bit too much detail and makes it difficult for different parts like hills and trenches to be recognised when looking from a perspective view. This could be countered by increasing the smooth value or reducing the amount of persistence (see Figure 35).

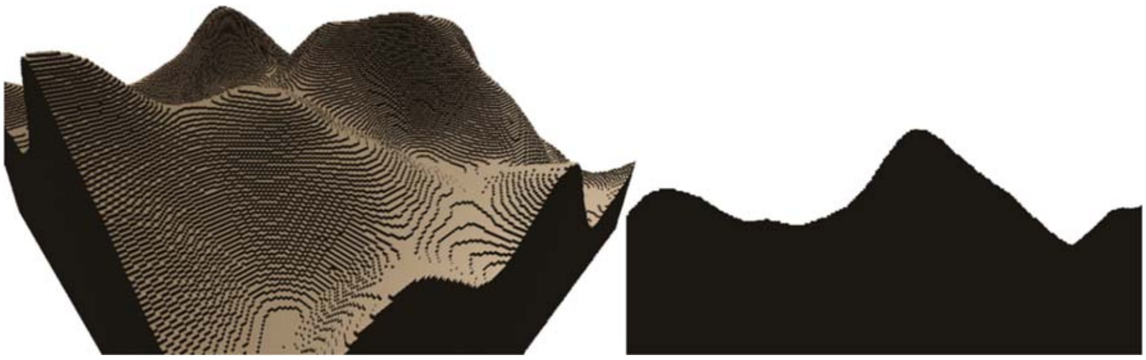


Figure 35: Achieving similar effect to 1 and 2 octaves by reducing the value of persistence between layers to 0.2 from 0.5.

Even though Figure 35 makes the terrain look slightly better than Figures 31 and 32, its cost is 8x and 4x bigger respectively so it is not worth generating.

After the visual comparison, the author believes that 4 is a good number of octaves to generate great-looking terrain. Even though a similar effect can be achieved with more octaves by fiddling with the smoothness and persistence values, 4 octaves create a good medium point where the terrain has enough detail and makes it easy to differentiate the various hills. Of course, the result depends on the observer’s view and the game that needs to be created. There are games with the type of art which could easily use only 1 or 2 layers of Fractal Brownian Motion.

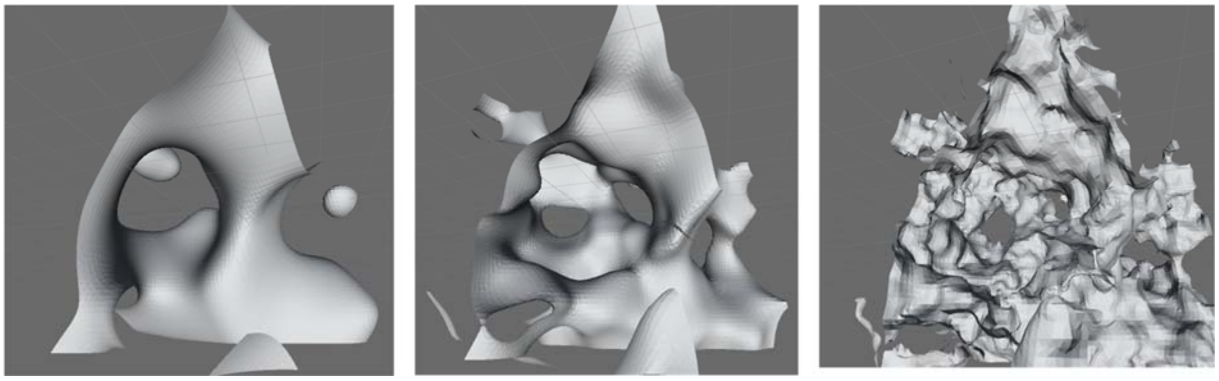


Figure 36: Marching Cubes with Fractal Brownian Motion.

On Figure 36: An example of 1,2,4 octaves (from left to right respectively) that can be used to achieve different type of complexity of a cave system when applying Marching Cubes on Fractal Brownian Motion generated terrain data. The three images are on the same size terrain and represent different levels of detail.

Project Outcome

Overall, the results of this evaluation were very satisfying and a lot of diversity in performance and visual satisfaction was achieved which marks the research quite successful from the author's point of view. The optimisation techniques were quite successful and made it possible to generate an endless world which can be visually satisfying and used both in types of games that are voxelised, as well as realistic ones (by extracting terrain meshes with the Marching Cubes algorithm). The paper elaborates more on this in the next section that validates the "Aim and Objectives Validation" as well as the next chapter "Conclusion" that talks in-depth about the project's success and its future.

4.4 Aim and Objectives Validation

The aim of this research paper was to analyse and implement techniques used in procedural voxel terrain generation. In order to achieve the aim, the following objectives were set:

Investigate static, dynamic or a combination of both techniques for procedural voxel terrain generation and choose one of the approaches.

Static and dynamic techniques were investigated as well as optimisations for their performance were introduced. The final tool implements a mixture of both ways of generation as an optimal solution. When the game world gets created, two iterations are done: first one creates the world statically and the next one dynamically. Depending on the values that are given to the tool and a game's requirements, the generation can be shifted to either be more static or dynamic-heavy. Thanks to the Recursive Flood Fill algorithm, a lot was learned about multithreading.

Investigate methods of procedurally generating terrain data and pick two for further analysis, understanding and implementation.

This objective was initially aimed at comparing Perlin and Simplex noise only but after deeper research and a discussion with the supervisor was done, it was discovered that this would not be challenging enough and that some other topics could be discovered. This lead to the opportunity to study Fractal Brownian Motion, arching Cubes and Volume Ray Casting techniques. The former two were successfully implemented and the latter one was theoretically researched and added to the potential list of future implementations.

Implement the procedural terrain generation and render terrain.

This was the practical objective that built upon the previous two and due to the popularity of procedural generation and available information and resources online, this was successfully implemented as a useful tool to use when creating procedural worlds for games or simulations.

Optimise and evaluate the performance of rendering terrain using 2 different techniques.
Both optimisation and evaluation were performed in various ways shown earlier in this chapter. Optimisation was the biggest aspect of creating, deleting and storing the terrain data and a few iterations were done. Evaluation was done on all the algorithms by exploring two scenarios where different variables of the algorithms were adjusted to test the different noise functions against each other as well as themselves when considering two factors – performance and visual appearance.

Gain knowledge in procedural content generation for future projects.
As the author's personal reason, this was one of the most important, as it has been stated throughout the paper, and a lot of theoretical and practical knowledge was gained to be able to use in a project aimed at creating a complete game world with different procedurally generated terrain and biomes.

4.5 Summary

This chapter presented the outcomes of this paper. It talked about the metrics taken during the evaluation phase of the implemented techniques; the results of carrying out the comparison of noise functions and the different optimisation stages; the two scenarios of evaluation were presented; screenshots from the tool's generated terrain were shown. The chapter concluded with a validation of the aims and objectives of the paper.

Chapter 5: Conclusion

This chapter shows a summary and author's reflection upon the project, its implementation, what went well, what didn't go as good, as well as how this project will be built upon in the future.

5.1 General Summary

Video games are an industry that is growing exponentially and we have only begun tapping into its limitless potential. The demand of non-repetitive experience for players makes for an enormous field to research in – procedural generation. As such a vast topic, many aspects of procedural generation were explored. Two noise functions (Perlin and Simplex noise) were successfully compared and their performance and visual impact was evaluated and presented. The outstanding realism in landscapes that Fractal Brownian Motion can achieve was discussed as well. Recursive Flood Fill algorithm was implemented to help making the terrain generation dynamic. Marching Cubes algorithm was used as the technique for extracting mesh and creating a realistic cave system from the voxel terrain data generated with the Perlin noise. Volume ray casting was investigated theoretically but not implemented due to project's focus on only comparing CPU-based techniques. Two scenarios for evaluation were designed – a scenario where the noise functions were compared together with Marching Cubes algorithm; and a scenario where the focus was on the loading and destroying of terrain data. A tool to generate infinite terrain data was created. This tool can serve as the beginning of development for any game that can fit non-repetitive experience in its core mechanics.

5.2 What Went Well

The author's belief is that the project was very successful. Thanks to it, a lot of knowledge on the topic was gained, a handy tool was implemented and good techniques for procedural generation were found.

5.3 What Didn't Go So Well

Procedural generation is a vast area of development and while some techniques like noise were deeply explored, others were only touched at the surface, like Volume Ray Casting.

One of the issues faced was related to the version of Unity used which was too old to take advantage of a deeper multithreading investigation. If a newer version of Unity was used, it could have been able to target a .NET version that comes with multithreaded data structures that are well-optimised instead of having to manually implement them. This could have led to two gains – faster dynamic terrain generation and time spent looking into improvement of the tool by further investigating another topic to implement. This was discovered quite late in the project and would have been difficult to upgrade the whole tool to address the issue. In the future, the tool could be brought up to newer Unity versions which use .NET's multithreading classes or the new Unity ECS (Entity Component System) which is considered extremely performant, especially when it comes to multithreaded code.

5.4 What Comes Next

Due to the large amount of area that procedural generation covers, it is possible to choose from many areas in which to look deeper in order to gain more knowledge. Multithreading and improving efficiency of data generation is definitely one step to investigate further in order to allow for faster-paced games.

Another area which the author found quite interesting and would love to extend the tool in, was the Marching Cubes algorithm. There are many well-known algorithms like marching tetrahedra, asymptotic decider, dual contouring, which serve as improvements of the flaws of Marching cubes algorithm of sometimes creating inconsistent topology of the mesh that is extracted from the scalar fields.

Yet another area to look at, as mentioned in earlier chapters, is transferring some of the computations onto the GPU and investigating whether the mix of CPU and GPU work would outperform the current CPU-only implementation.

Last but not least, the tool could be used in the future to create a game with different biomes based on factors like height of terrain, latitude and longitude, humidity, season, and others.

Chapter 6: References

All references are added in order of appearance unless they represent a research paper.

[1] “Investing in the Soaring Popularity of Gaming”, Reuters, Available at: https://www.reuters.com/sponsored/article/popularity-of-gaming

[2] ACM Digital Library, Available at: https://dl.acm.org/dl.cfm

[3] Google Scholar, Available at: https://scholar.google.com

[4] “Volume Pixel (Volume Pixel or Voxel)”, Technopedia, Available at: https://www.techopedia.com/definition/2055/volume-pixel-volume-pixel-or-voxel

[5] “A series of voxels in a stack with a single voxel shaded”, Wikipedia, Available at: https://en.wikipedia.org/wiki/Voxel#/media/File:Voxels.svg

[6] “Voxel model for Staxel game”, AmazeTNT at ArtStation, Available at: https://amazetnt.artstation.com/projects/19LkK

[7] “Illustration of a voxel grid, each containing a color value”, Wikipedia, Available at: https://en.wikipedia.org/wiki/Voxel#/media/File:Voxelgitter.png

[8] “CryEngine 3: A Comprehensive Introduction”, 3DBuzz, Available at: https://www.3dbuzz.com/training/view/introduction-to-cryengine-3/details

[9] “Making it in Unreal: how voxel magic makes the world burn in Firefighting Simulator”, Se7en, Available at: https://se7en.ws/making-it-in-unreal-how-voxel-magic-makes-the-world-burn-in-firefighting-simulator/?lang=en

[10] “Terrain”, Thinglink, Available at: https://www.thinglink.com/scene/854712290216247298

[11] “Procedural Generation”, Wikipedia, Available at: https://en.wikipedia.org/wiki/Procedural_generation

[12] “No Man’s Model Viewer”, No Man’s Sky Mods, Available at: https://nomansskymods.com/mods/no-mans-model-viewer/

[13] “Dynamic vs Static Procedural Generation”, Medium, Available at: https://medium.com/@eigenbom/dynamic-vs-static-procedural-generation-ed3e7a7a68a3

[14] “Far Lands”, Minecraft Gamepedia, Available at: https://minecraft.gamepedia.com/Far_Lands

[15] “Value Noise and Procedural Patterns: Part 1”, Scratchapixel 2.0, Available at: https://www.scratchapixel.com/lessons/procedural-generation-virtual-worlds/procedural-patterns-noise-part-1/introduction

[16] “Noise”, The Book of Shaders, Available at: https://thebookofshaders.com/11/

[17] “Gradient Noise”, Wikipedia, Available at: https://en.wikipedia.org/wiki/Gradient_noise

[18] Perlin, K. (2002). Improving noise, San Antonio, ACM SIGGRAPH Computer Graphics, 681-682

[19] “Simplex noise”, Wikipedia, Available at: https://en.wikipedia.org/wiki/Simplex_noise

[20] Lorensen, E.W. and Cline, E.H. (1987). Marching cubes: A high resolution 3D surface construction algorithm, New York, ACM SIGGRAPH Computer Graphics, 163-169

[21] “Rendering (computer graphics)”, Wikipedia, Available at: https://en.wikipedia.org/wiki/Rendering_(computer_graphics)

[22] “Ryse Polygon Count Comparison with Other AAA Titles – Star Citizen, Crysis 3 and More”, wccftech, Available at: https://wccftech.com/ryse-polygon-count-comparision-aaa-titles-crysis-star-citizen/

[23] “Overworld”, Minecraft Gamepedia, Available at: https://minecraft.gamepedia.com/Overworld

[24] “Chunk”, Miecraft Gamepedia, Available at: https://minecraft.gamepedia.com/Chunk

[25] “Multithreading (computer architecture), Wikipedia, Available at at https://en.wikipedia.org/wiki/Multithreading_(computer_architecture)

[26] “ConcurrentDictionary at GitHub”, João Parreira, Available at: https://github.com/realtime-framework/unity3d-plugin/blob/master/Messaging/Internal/ConcurrentDictionary.cs

[27] “ConcurrentDictionary<TKey,TValue> Class”, Microsoft, Available at: https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentdictionary-2?view=netframework-4.7.2

[28] "Perfect Pitches with a Rubber Band Guitar”, Science Buddies, Available at: https://www.sciencebuddies.org/teacher-resources/lesson-plans/sound-wave-frequency-amplitude

[29] “Fractal Brownian Motion”, The Book of Shaders, Available at: https://thebookofshaders.com/13/

[30] “Marching cubes”, Wikipedia, Available at: https://en.wikipedia.org/wiki/Marching_cubes

[31] “Contour line”, Wikipedia, Available at: https://en.wikipedia.org/wiki/Contour_line

[32] “Raymarching Distance Fields: Concepts and Implementation in Unity”, Adrian’s soapbox, Available at: http://flafla2.github.io/2016/10/01/raymarching.html

[33] “Volume ray casting”, Wikipedia, Available at: https://en.wikipedia.org/wiki/Volume_ray_casting

[34] “Mali GPU Application Optimization Guide”, ARM Software development tools, Available at: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0555a/BEIGDEGC.html

[35] “Rendering Statistics Window”, Unity documentation, Available at: https://docs.unity3d.com/Manual/RenderingStatistics.html

[36] “Single threaded dynamic generation of terrain”, Stefan Kopanarov, Available at: https://youtu.be/ywHMr8vscqA

[37] “Marching cubes terrain”, IndieDB, Available at: https://www.indiedb.com/games/khora/videos/marching-cubes-terrain

[38] Perlin, K. (1985). An image synthesizer, San Francisco, ACM SIGGRAPH Computer Graphics, 287-296

[39] Gustavson, S. (2005). Simplex noise demystified, Linköping