Performance analysis of terrain generation techniques for real-time simulations.

MComp Computer Science w/ Industrial Placement (Game Engineering)

Supervisor: Dr. Graham Morgan

May 2017

Word Count: 14,894

Jack Lloyd (130367062)

Abstract

This dissertation researches the effectiveness of different procedural terrain generation techniques running within a real-time simulation, to generate real-looking/natural terrain for video games.

A summary of the subject area is presented; the design decision and development of a real-time terrain generation tool is documented. The authors hypothesis being that Simplex Noise is the best technique to generate natural looking terrain on an infinite scale within a real-time simulation, is tested by an evaluation of the computer hardware's data during runtime. The results and analysis are presented, along with additional work that could be carried out in the future to further this project.

Declaration

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

Acknowledgements

Firstly, a thank you is due to the supervisor of the project Dr. Graham Morgan, for allowing this project to be undertaken. He has helped with the organisation and provided continued advice throughout. Thanks also are given to Dr. Gary Ushaw and Dr. William Blewitt, who have also provided help and guidance throughout.

Table of Contents

Table of Figures

Introduction.

Design and Implementation

Development

Table of Code

Development

Evaluation

Table of tables

Evaluation

Chapter 1: Introduction

This section aims to introduce the reader to the project, providing the purpose of the project along with the aims and objectives. Throughout this project there will be many abbreviations and terminology therefore, included is a glossary which can be found at the back of this document.

1.1 Subject Area

Video games are a large part of today's world with no signs of slowing down, from handheld devices to dedicated machines in our homes, and this trend is predicted to continue until at least 2020. With this trend, developers are pushing out more titles, each attempting to one-up the previous.

Consumers are as ever expecting top of the line products, after all, there are laws in place to avoid the misleading of consumers and protect them. Each year consumers expect more and more. There is nothing different with the game industry, smoother, better looking, more frames, more realistic are terms found across the industry, with this of course comes terms such as more spending, more hiring, more development time, from a studio point of view, which in turn means higher prices for the consumer. Bad.

How can studios offer more game in a game without increasing the cost, as to make the consumer happy? Enter procedural content creation, enabling the game to generate aspects of the game itself.

1.2 Purpose

As the video game industry continues to grow both in terms of output and audience and is projected to continue this trajectory until 2020+ [1] the more game studios will need to rely on the use of technology in order, to increase their output, to meet consumer demands.

A prime example of this is terrain. Terrain is a game asset. It is present in arguably every video game, from a simple scene in which just a box is present on the ground to a complex and spectacular landscape mountain scene, this means that terrain within a video game is a major part of the game, it's important. It's even more important if that terrain is a going to be a selling point.

Figure 1: Landscape scene from Skyrim, Bethesda. [2]

The image above is of Skyrim the latest addition to the Elder Scrolls franchise from Bethesda. It features a large open world with undeniably beautiful landscapes Figure 1. The issue? This is very expensive for Bethesda. The process of creating any asset within a video game is expensive, due to the sheer number of professional 3D artists needed and the time scale associated with asset creation, required to achieve the high level of detail that everyone is seeking.

This is where procedural content generation comes into play. Procedural content generation is the process of allowing the software itself to generate parts it needs. Procedural systems aim to lower the cost of creation.

> *"The main reason to create content procedurally is that it can be cheap. Really cheap. Generating new content is a matter of turning knobs and moving sliders. At that point, our imagination is the limit of what we can do, and there's very little work involved in creating each asset."* [3]

What does this mean in terms of video games? No longer are 3D artists needed. No longer do assets take hours upon hours to design and hours more redesigning.

Procedural terrain generation is when a game engine generates the terrain in runtime. This means every load of the game has a different world in which the player interacts. For a sense of infinite possibilities.

This dissertation looks at the algorithms used in the video game industry to achieve the above.

1.3 Project Aims and Objectives

This dissertation aims to evaluate different techniques used to generate terrain procedurally in real-time simulations.

This aim has been divided into the objectives below:

1. To identify generation techniques used by industry today.

This objective includes, research into the different terrain generation techniques that are used and identifying what techniques this dissertation will be addressing.

2. To develop a tool, in which the techniques can be ran and compared.

This step is to ensure that during the evaluation phase only the technique used to generate the terrain is changed. Ensuring that the drawing procedures, game logic and player movement remain the same throughout. This objective is important as failure at this stage would mean the results would be inconsistent, an example of this would be change to the FOV would mean more terrain has to be rendered on screen at any given time, giving the GPU more work to do, when compare to a lower FOV value. The tool will allow the real-time switching and visualisation of the technique being used. It will display relevant information on screen such as the current technique, world space location of the camera and memory usage.

3. Implement the techniques into the tool.

Integrate the different techniques identified in objective 1 into the tool developed in objective 2. Ensuring the underlying framework stays the same between techniques as outlined previously.

4. Evaluate the techniques.

The dissertation will include the findings from the CPU, GPU and RAM usage, including a look at the FPS count. Min, max and averages achieved, and importantly the overall visual representation of the technique. After all this for use in video games which are pushing the boundaries on the appearance of real-time simulations after all.

> *"Recreating a Mission Impossible experience in gaming is easy; recreating emotions in Brokeback Mountain is going to be tough ... it will be very hard to create very deep emotions like sadness or love, things that drive the movies"*
>
> *"Until games are photorealistic, it'll be very hard to open up to new genres. We can really only focus on action and shooter titles; those are suitable for consoles now."* [4]

Chapter 2: Background Research

This chapter addresses the research carried out for this dissertation. Real-time simulations are looked at, procedural generation and the concept of noise functions are introduced.

2.1 Research Strategy

Research undertaken was conducted using both proven and informal sources

Proven publication research

- Research papers on noise functions and procedural techniques were found through the use of google scholar and the university library including the use of libsearch.ncl.ac.uk
- Magazine research came from reading various editions of the Game Developer Magazine which can all be found online under the computer archives section of GDC vault.

Informal research

- Forums used include sites such as gamedev.net and se7ensins.com. Where posts regarding noise functions and implementation solutions were discussed.
- Web sites other than forums have also been used, these includes sites that come under the category of personal websites of software engineers.

2.2 Background Research

2.2.1 Real-Time Simulations

What is a real-time simulation and why do we use them?

Simulation tools are a quick and cost-effective method of prototyping a design and verifying it acts as expected before having to press that dreaded print button. Simulation tools over the years have progressed alongside the progression of computing power. More Powerful Computers = More Complex Simulations

> *A simulation is the representation of an environment. They have been widely used in the electrical industry in the design phase of electric circuits.* [5]

Working out what connections are needed where how much current or what the voltage is at a given time. Simulations play a crucial part in the development of systems.

Up until the last couple of years there has existed a saying that overall processing power for computers will double every 18 months, this has been because the number of transistors on a chip has been increasing. In an almost identical pattern simulation tools have progressed.

As computer power, has increased the capabilities of simulations to carry out complex simulations in shorter time has grown. Enter real-time.

Real-time simulations as the name suggests are simulations that report back data and react to changes in real time, as said changes occur. A simulation can be running and inputted data during the running time. Not reliant on pre-determined values for variables for example.

Why is this important? Well a video game is just a real-time simulation. An environment “world” is simulated and the user “player” can change the state of the simulation at any given time returning the results of the simulation in the form of an updated world on screen.

2.2.2 Procedural Generation

Here we discuss what procedural generation is and how it is important to video games.

As previously touched on procedural generation is the process of allowing the software itself to generate parts it needs, with the aim of lowering the cost of creation in terms of man hours spent on designing it.

> *“So, what’s the point of Procedural generation? The main problem of game development today is not lack or ram or storage, it is the cost of filling it with content.”* [6]

You should remember that the hardware video games that are running on today is a lot more powerful than the last generation, not to mention the generation before that or even the one before that one. The biggest issue was memory and this was usually the limiting factor for a game, however this has now changed. Therefore, memory or the lack of memory is no longer the problem, it’s now about how much it cost the game studio to fill that memory with content.

Game studios today use procedural generation in a number of ways. The original Diablo uses the technique to randomise the dungeon layouts and the item generation [7]. Borderlands uses it to generate the weapons avaible to the player [8]. Spore uses the procedural system for the animations [9].

It’s important to distinguish between the different types of procedural generated content.

2.2.2.1 Online or Offline

There’s one distinction which needs to be addressed first, that’s the two types of procedural generation that exists. The first is known as offline mode. First a piece of art known as the sample is generated by an professional artist, this is then used to kick-start the procedural generation. This can be seen in Figure 2 and Figure 3: Models generated from the base model in Figure 2. . The recently released controversial No Man’s Sky game uses assets known as base models, from this base model a theoretically infinite possible models can be procedurally generated.

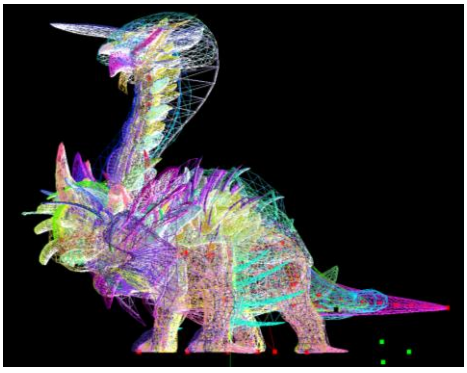


Figure 2: No Man's Sky Triceratops model. [7]

Figure 3: Models generated from the base model in Figure 2. [10]

The second type is online mode, where the generation is all done in real-time whilst the system is running. Has no artist is involvement. In a game example, this may be as often as every frame as some in-game asset may have the chance to change its mesh state that often, i.e. new terrain in Minecraft needs to be generated as the player moves around for the first time.

Figure 4: Landscape scene from Minecraft, Mojang. [11]

This dissertation is going to investigate both sides of the story, offline mode and online mode. Looking at various techniques of procedural generation, specifically focusing on how terrain generation can be achieved to create a game that has infinite terrain. A prime example of a game which uses this affect is Minecraft as seen above in Figure 4.

Offline Heightmap generated terrain.

The first technique to be addressed is of an offline type, as mentioned offline types use a pre-made piece of art to generate the procedural content needed. In the tool developed, this technique is showcased by using a Heightmap. This Heightmap is then used to generated the terrain that is being rendered by the nclgl (Newcastle University Graphics Library from 2015) framework. A Heightmap is a text file, which stores raw values, these can be used as the Y component directly.

Online Midpoint Displacement generated terrain.

The second technique is a midpoint displacement algorithm. This is an online type of procedural generation. Unlike the first technique no data is needed prior. No artwork is produced by an artist to kick-start to process. This technique is also known as The Diamond Square. This is due to the nature of the algorithm employed.

Once the initialisation has been done the algorithm follows the following pattern, perform a diamond step which sets the midpoint of the square based on the average of the four corner points plus a small random value, this is followed by a square step which sets the midpoint of the diamond to the averages of the four corners and another random value.

These steps are repeated until all the map is filled with values. The map being an array for instance. This array now holds the Y part of X, Y, Z coordinates and the nclgl framework can use it to render terrain. Each X, Z location has a map to a Y value.

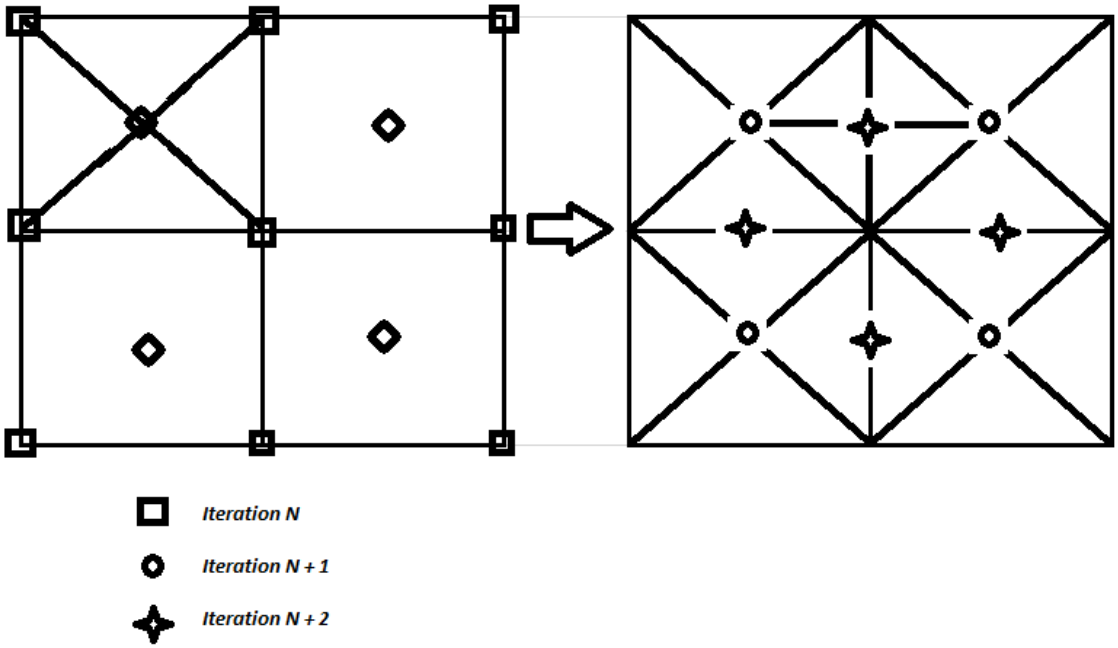


Figure 5: Visualisation of the Diamond-Square steps, recreated personally. [12]

Online Noise Algorithmic generated terrain.

The final techniques which will be looked at in this dissertation are procedural generated content from noise functions. As with the diamond square technique above this does not need any art to be made prior to the generation, and again afterward each X, Z location maps to a Y value. The difference is how this Y value is generated.

In these techniques, the Y value is calculated by using what's known as noise functions (discussed below) for every X, Z value and not the averages of three or four previously known values as in The Diamond Square technique.

2.2.3 Noise Generation

> *"Noise is a series of random values, typically arranged in a line or a grid. In old TV sets, if you tuned to a channel that didn't have a station, you'd see random black and white dots on the screen. That's noise (from outer space!). On radios, if you tune to a channel that doesn't have a station, you hear noise"* [13]

Ask the majority and you'll hear that noise is generally unwanted. If you're trying to hear someone in a noisy environment, you'll be wishing it was quiet. Less noisy. In electrical circuits noise is unwanted. On TV noise is unwanted. On radio noise is unwanted. Get the picture? Noise like this could be left over background radiation that is being picked up by the circuit components or interference from close by devices. Thus, you tend to want to remove any noise or as much noise as possible from your data.

> *"in most applications, you're trying to subtract the noise"* [13]

However, whilst looking at natural systems such as the landscapes (terrain on earth). You can see that they look noisy.

Figure 6: Mountain range, showcasing earths noise. [14]

So therefore, when trying to procedurally generate terrain, as it's a natural system, noise should to be added, rather than subtracted. Figure 6 shows this, the mountains range if drawn in 1-dimension would be very noisy as seen here.

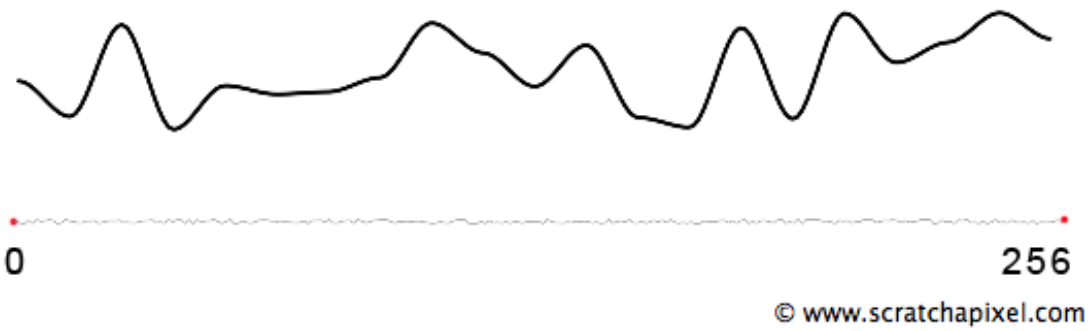


Figure 7: Output from a 1D noise function. [15]

The similarity between the mountainous region and the noise function output, show that noise functions are good solutions to generating procedural terrain.

> *"If it is possible to list all the properties that an ideal noise should have, not every implementation of the noise function matches them all (and quite a few versions exist).*
> *Noise is pseudo-random and this is probably its main property. It looks random but is deterministic. Given the same input, it always returns the same value. If you render an image for a film several times, you want to noise pattern to be consistent/predictable. Or if you apply this noise pattern to a plane for example, you want that pattern to stay the same from frame to frame (even if the camera moves or the plane is transformed. The pattern sticks to the plane. We say that the function is invariant: it doesn't change under transformations)."* [15]

As touched on earlier, there are several noise functions even so, they have things in common. No matter what dimension the noise function is operating in, 1D, 2D, etc. the function will return a float value, and they are pseudo-random, that being the randomness is deterministic, meaning that for the same input set A, the same output set B will be returned, every time. 1D, 2D? This is how many parameters the noise functions take. 1D takes 1 parameter as the input, 2D takes 2 etc.

Commonly 1D functions are used for animation purposes, 2D and 3D are used for texturing and terrain generation. Higher dimensions are used for obscure simulations.

> "1D noise is used for animating objects, 2D and 3D noise are used for texturing objects. 3D noise is particularly useful for modulating the density of volumes." [15] [16]

Noise functions

Perlin Noise

Perlin noise is a lattice-based (grid approached) gradient noise type. Perlin noise was developed by Ken Perlin (a professor in the Department of Computer Science at New York University) in 1983 and was the first gradient noise implementation.

Grass, grain, wood, marble, clouds are all natural and when modeling them you don't want a repeating/flat or perfect texture you want some imperfections to add that natural look to your scene. Perlin noise is good for this. Perlin noise also conforms to the pseudo-random (PRN) mentioned above.

> *"The essential fact for us is that Perlin Noise is an application from Rn to R, that is, for each point in an n-dimensional hyperspace, PNG (x1, x2, ..., xn) returns the evaluation of the Perlin Noise primitive in that point. Moreover, the PNG is consistent, meaning that, if (x1, x2, ..., xn) = (y1, y2, ..., yn) then PNG (x1, x2, ..., xn) = PNG (y1, y2, ..., yn),"* [17]

Simplex Noise

Simplex noise is comparable to Perlin noise mentioned above but the generated noise pattern has fewer artefacts (errors) at the same time having a lower computational cost in higher dimensions. It was again developed by Ken Perlin, in 2001 as to address the limitations of noise functions in the higher dimensions.

Whereas the big-O of Perlin noise due to the limitations of the implementation is exponential at $O(n*2^n)$ Simplex Noise has a big-O of $O(n^2)$.

> *"Therefore, the computational complexity of previous Noise implementations in n dimensions is O(n2n), whereas the computational complexity of the new Noise implementation, in n dimensions is O(n2)."* [16]

Further distinction between the two comes in the form of Simplex Noise being patented when used in 3D or above and for texturing purposes.

Value Noise

Noise based on values. Value noise unlike Perlin and Simplex mentioned previously is not a gradient type, but it is still a lattice approach. Given a PRN at each grid point a new value can be interpolated (a method of generating new data point within the range of a discrete known points) from the values instead of the gradient vectors being used.

One downside to using a value based noise function is that the frequency of close values is higher and that is that you may find that the outputs are very close for a longer period of time than that of a gradient approach.

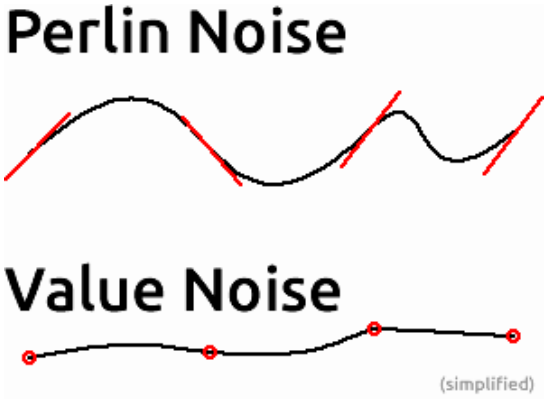


Figure 8: Value vs Perlin 1D output. [18]

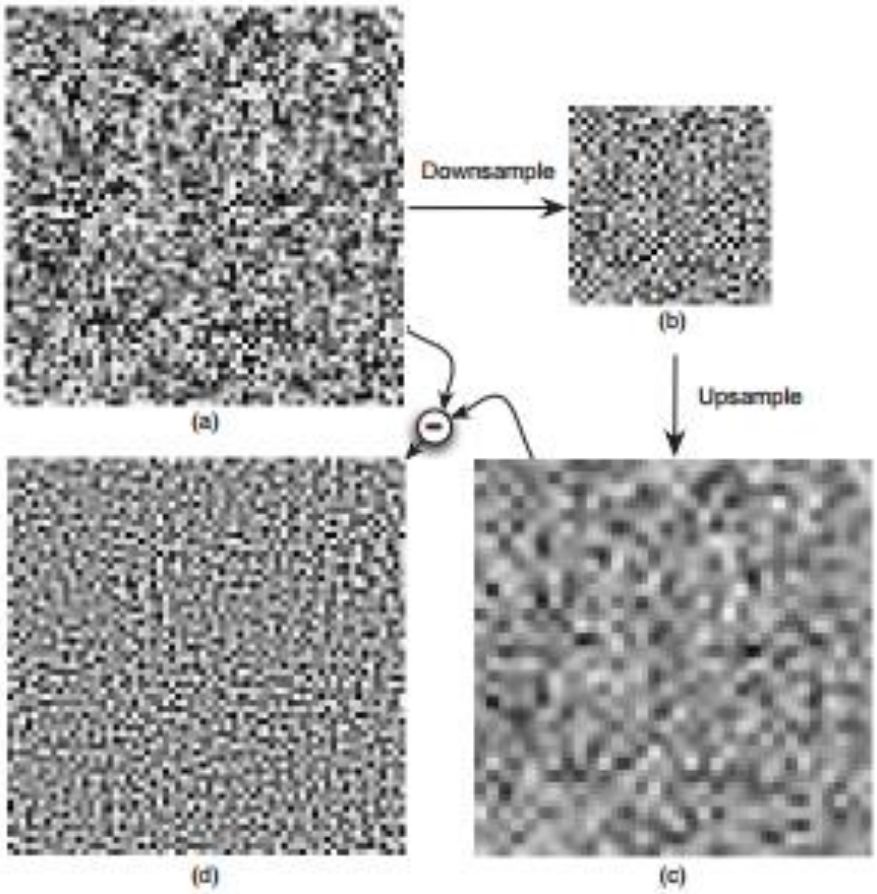


Figure 9: Wavelet Noise. Random values, down-sample, up-sample, subtraction. [16]

Wavelet Noise

Wavelet whilst born due to problems with detail loss associated with Perlin Noise, Wavelet Noise is not a gradient type or even a value type. Wavelet is an explicit type meaning the noise is pre-processed and stored for access later. Wavelet is generated by down-sampling and then up-sampling a grid of random values. The original is then subtracted from the upscaled values, producing the noise. [19]

2.3 Summary

With the increase in computer power over the last 2 decades the complexity of real-time simulations has increased alongside it. A video game is just a real-time simulation. We are simulating a world for the player to interact with. With the advancements in computer hardware no longer are game developers limited by the hardware. No longer does the amount of RAM come into play. No longer are developers worried that a texture file might be too big, take Fallout 4 it just got a 58Gb texture pack. 58GB! Instead the question being asked, if why are we not using all the memory we have? And can what can we do to fill all the memory we have? because of this the reasoning behind procedural content has changed.

No longer are they used to overcome the limitations of storing textures for example, instead they are being used to create different gameplay experiences for all the players each and every time they play. Minecraft, that game is a perfect example, it has worlds which are different (assuming no seed has inputted) every time. How is this done. Procedurally generated content, i.e. the terrain.

Chapter 3: Design and Implementation

3.1 Introduction

This section details what work was done for this project and why it was done. This section will address the design decisions and why they were chosen, listing advantages and disadvantages along with each justification. The final implementation is discussed from start to finish, what tools and technologies have been used in order to achieve the final implementation of the tool developed for this project.

3.2 Planning

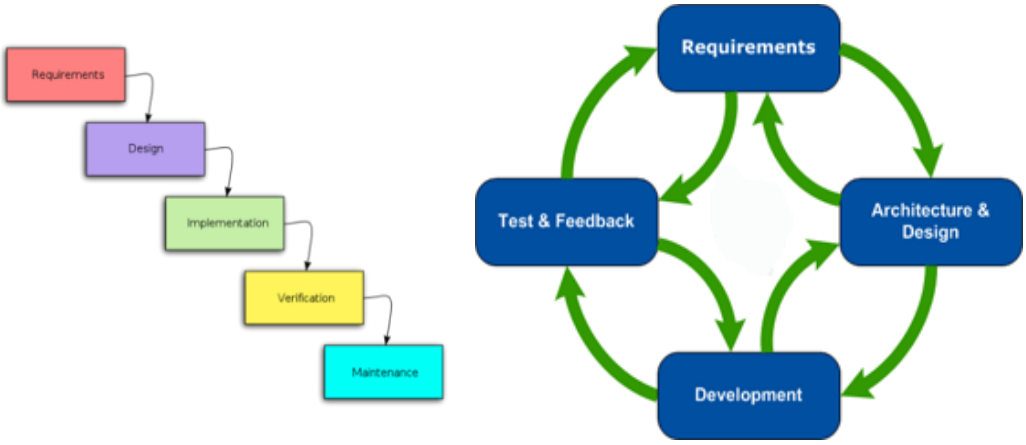


Figure 10: Waterfall model (left) | Agile model (right). [20]

From the start, due to the nature of the project and how regularly meetings with supervisors was going to be an agile model was adopted. Agile in a software development environment is far superior to the traditional waterfall method.

Waterfall has no protection when it comes to poor quality, poor visibility, risk and change. It treats each stage of the software lifecycle as a discrete phase therefore, no backward tracking is possible.

What happens if a project runs out of time, well it still needs to be shipped so let's scrap testing.

Waterfall model means no software is developed until the second half of the life cycle, meaning you never really know at what development stage the project actually is. It does not handle change.

The agile approach fixes all these issues outlined, by treating each phase concurrently with the other phases.

As shown in Figure 11 (shown below) instead of the entire software development being treated as one big activity, you can imagine it as each sub-section of the development is treated as its own activity. With each step being worked on concurrently. Coding is not left until the end and testing is done continuously. Quality improves, testing begins early. Reduced risk as feedback starts early. Easier to adapt to changes from the customer.

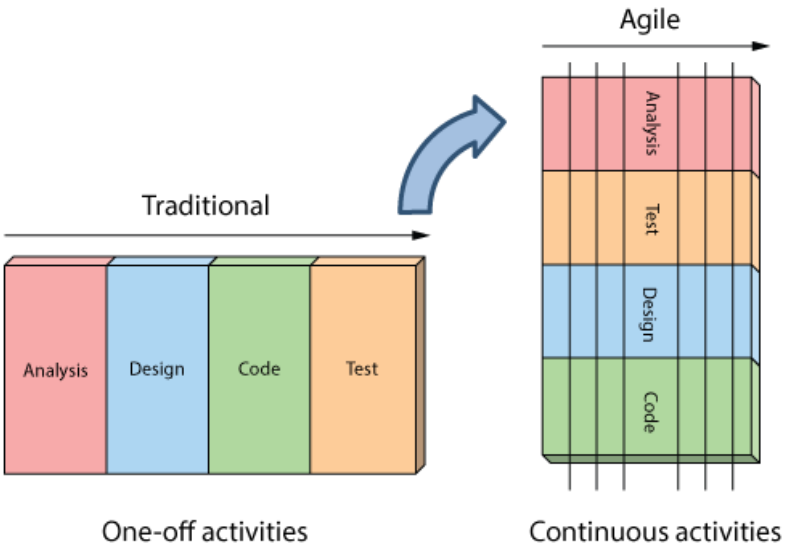


Figure 11: Difference between Waterfall and Agile approach. [21]

This method fitted this dissertation perfectly. At first weekly meeting were held with the projects supervisor. These were to support the project, any help required was available, whether that be related to the dissertation direction or help with software development.

What techniques to be implemented was the next decision to make. This dissertation from the start had a clear goal of wanting to demonstrate the key differences between the chosen techniques outlining where these could be used and why, with this is mind it was obviously to pick a good number of techniques whilst keeping it entertaining and enjoyable. End goal was to narrow down on which technique is best for terrain generation.

After the research carried out above it was decide that in order to showcase the best techniques the worst ones would also need to be included in the tool. Enter heightmap. Now it's not to say that the simplest technique is the worse, but going into the project from the research we knew that this technique would be very limited yet simple to implement as it reads values directly from a piece of art. [2.2.2.1 Online or Offline - Offline Heightmap generated terrain.]

Secondly from researching it was decided that a middle man approach was needed. Not a full offline method but nothing associated with noise functions either. Diamond Square. This algorithm is simple and yet is shown to produced good natural looking terrain, shown below in Figure 12.

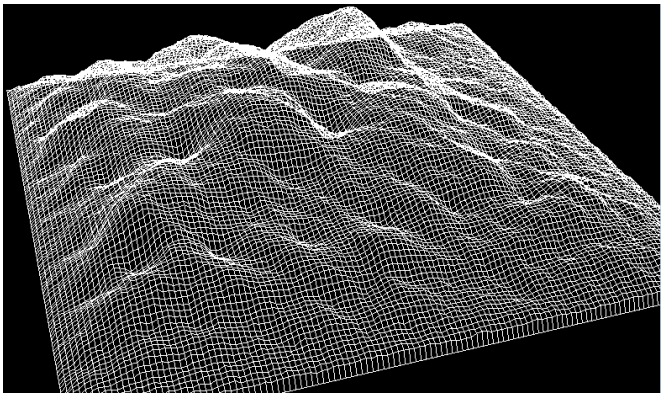


Figure 12: Outcome of one Diamond-Square algorithm. [22]

After addressing two non-noise techniques it was decided to look at the noise functions. Simplex noise was a must have, since it is the most recent implementation released from Ken Perlin, speaking of which Simplex's younger brother is also needed Perlin. As stated in [2.2.3 Noise Generation - Perlin Noise] Perlin Noise was the first type of gradient noise. The final approach was a little trickier to narrow down. The tool needed a noise function which was different yet fundamentally the same as Perlin and Simplex. This is where Value Noise comes in. Value noise as touched on above [2.2.3 Noise Generation - Value Noise] is different from Simplex and Perlin in that it uses the pure values of the grid points instead of gradient vectors. Which should mean because of the higher chance of close values [Figure 8: Value vs Perlin 1D output.] the terrain is more flowing.

3.3 Tools and Technologies

This section is designed to introduce the tools and technologies that are being used in the final end tool developed. The IDE (Integrated Development Environment) will be address along with why this language was used and what API's have been utilised.

3.3.1 C++

C++ is a general purpose object-oriented programming (OOP) language. It's an extension of the C language and it's considered as an intermediate language. That is that it has both high and low-level features.

The OOP aspect or the addition of classes into the C language is the main reason for C++.

Firstly, unlike other languages such as Java (that is what is taught for the first 2 stages here at Newcastle University) C++ runs directly on the hardware not in a virtual machine. This means that C++ offers more control over memory. You can then have the ability to say where you want some memory and what exactly to do with that memory. In games this is crucial. Imagine a java based game, every few frames the built-in garbage collector ("The garbage collector checks every piece of memory you allocated every x frames and determines whether it's garbage or still being used." [23]) runs. That is going to be a huge slowdown for your game.

Secondly, because of the above point many games are made using C++, because of this most of the libraries game developers want to use are written in C++. This includes the rendering library the developed software uses. [3.3.4 NCLGL].

Thirdly, to go along with the previously mentioned garbage collection, with C++ you only pay for what you want. No overhead costs. This makes it ideal for the high-performance needed from a real-time simulation, whilst maintaining that usability. Plenty of resources are available.

3.3.2 Visual Studio

Visual Studio was the IDE of choice for two reasons.

One it offers simple integration with GitHub (an online code repository). Prior knowledge of GitHub was limited and therefore a simple implementation within the IDE to carry out pull/push request. To ensure that a constant back-up was created.

The second reason for this IDE is that the nclgl rendering library was already a visual studio project, which made using it for the rendering a very easy task. No porting of the code was needed.

3.3.3 OpenGL

> *"OpenGL – The Industry Standard for High-Performance Graphics.*
> *Most Widely Adopted Graphics Standard.*
> *High Visual Quality and Performance.*
> *Developer-Driven Advantages.*
> *Well-documented."* [24]

Enough keywords? So, OpenGL (Open Graphics Library) is a well know graphics API (Application Programming Interface). It was first introduced in 1992 and has since gone on to become the go to API for rendering.

> *"Since its introduction in 1992, OpenGL has become the industries most widely used"* [24]

A major point regarding OpenGL is that it is not platform specific, meaning that an application will work across multiple different graphics cards.

> *"OpenGL specification is not platform-specific, it is possible to write an application that will be possible to use against many different types of graphics cards"* [24]

OpenGL is the bee's knees of graphics APIs, that is all it is. It handles rendering of graphics, nothing more, nothing less. It is not a API for I/O operation, animations or the GUI, etc. One feature of OpenGL that is key is that it frees the developer from having to design specific hardware features. The OpenGL drivers, enclose all the information about the underlying hardware. OpenGL is portable.

> *"All OpenGL applications produce consistent visual display results on any OpenGL API-compliant hardware"* [24]

3.3.4 NCLGL

Known as nclgl, Newcastle Game Library is Newcastle University's small game framework. This framework includes things such as a vector class, matrix class with all the necessary mathematical calculations already written.

> *"You also need to perform quite a lot of vector and matrix manipulation, which if coded incorrectly could cause very undesirable rendering results! To this end, for this module you're going to be provided with a small 'framework'"* [25]

This is important as this dissertation is in no way addressing how to do matrix manipulation.

NCLGL not only includes pre-made math classes like Vector2/3 and Matrix4 but also provides addition libraries such as SOIL a library to support the loading of textures, useful for possible future work.

It provides a windows class, which has all the code needed to create a Windows window. This includes such things as the Mouse interface, Keyboard interface and a GameTimer class.

> *"The Mouse class encapsulates all of the Windows mouse functionality provided via the RAW API. The Keyboard class encapsulates all of the Windows keyboard functionality provided via the RAW API."* [25]

3.3.5 FRAPS

Released in 1999 Fraps is a windows benchmarking tool. If the software is DirectX or OpenGL based then Fraps can be used to benchmark, record or capture the software screen output. This is application is going to be key during evaluation.

> *"Fraps is a real-world gaming benchmark that provides advanced frame rate reporting, screenshot capability, and gameplay capture all in a tiny, 2MB installation package. It's been evolving for years, faithfully delivering the dish on your system's gaming mettle."*
>
> *"Fraps is an easy recommendation for gamers seeking the hard numbers from their favourite titles. It isn't going to give you a mountain of data to consider. It'll just give you a single number – namely, your in-game FPS - but it is likely to be the exact answer you're looking for and all you really need."* [26]

That is exactly what was required, a hard figure for the FPS the developed tool can handle on each technique.

3.3.6 MSI Afterburner

> *"MSI Afterburner is the world's most recognized and widely used graphics card overclocking utility which gives you full control of your graphics cards. It also provides an incredibly detailed overview of your hardware …"* [27]

Detailed overview of the hardware, that's exactly what was needed. A consistent method of gathering data in regards to what the system was doing during each technique.

3.4 Application Specification

Before implementation could begin, what the application needed to be able to do, needed to be outlined.

Requirement 1 – Player controllable

- The user must be able to move around the world of their own accord, without issues.
- Full keyboard and mouse support to be included.

Requirement 2 – Infinite world

- The world in which the player can move, must be infinite in all directions.
- Starting at 0, 0

Requirement 3 – On the fly technique switching

- User must be able to switch the technique being used in real-time, no exciting required.
- All currently generated terrain will be re-generated using the chosen technique.

Requirement 4 – Easy to distinguish different height values

- To clearly show the differences between one technique to another, the tool must be able to clearly show the different height values of the terrain.
- Interpolation from full blue to full green, from 0 upwards.

Requirement 5 & 6 – Only the technique used changes & testing strategy must the consistent.

Other than being able to switch between the actual technique being used, requirement 5 is the most important. The testing / evaluation carried out would be invalid if things such as the rendering method, FOV (Field of View) or window size changed.

- Consistency between techniques other than the technique must be 100%.
- Rendering pipeline must stay the same.
- Shaders used must stay the same.
- Player / Camera movement during testing must be the same from one technique to another.
- Amount of terrain on screen at any time must be consistent between techniques.

3.5 Development

3.5.1 Introduction

This section will cover what development was undertaken to get the tool from conception to deployment. As stated previously, after deciding which techniques to evaluate, a framework to implement the techniques into was needed, see objective 2.

> '2 - To develop a tool, in which techniques can be run and compared.'

First a system to allow terrain to be rendered had to be created. The terrain in question was nothing special, a perfectly flat world, to be specific. The approach taken was to generate the world using a tiling system, like how Minecraft generates the world. Minecraft generates the world using a chuck system, a chunk is defined as 16*16 * 256 [11], the aim of this dissertation is not to generate a world in which as game can be played so a chuck system with where more than 1 Y value is needed would be overkill and unnecessary, a tile system is perfect. Each tile begins at a local coordinate of 0, 0 and ends at 256, 256 and holds a single Y value.

To achieve the above, the concept of index buffering must be introduced.

3.5.2 Indexing/Indices

Here we introduce how to use indices for geometry data, allowing access to a vertex multiple times in one draw cell. This is useful as the terrain generated will take up less space, whilst this is not important in terms of what we have available it's important as the program in the end will have a higher performance when compared to saving all the information about our terrain. Let us discuss this simple pyramid shape.

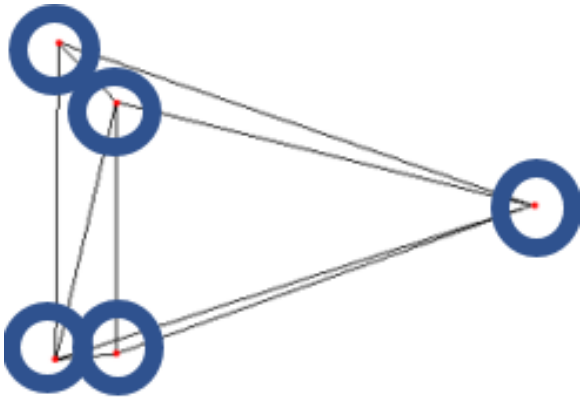


Figure 13: Simple pyramid w/ its vertices highlighted.

The circles represent the 5 unique vertices that the pyramid has, but in reality, the pyramid has 18 vertices if drawn using the GL_TRIANGLES type defined in OpenGL. 3 vertices for each of the sides and 4 for the bottom base. That's 18 vertices that need to be saved and yet 13 of them are identical to another. Does this information really need to be saved more than once? Enter indices.
As the name suggests you can imagine indices as a normal array, like an array can index locations OpenGL can index into the Vertex Buffer. This eliminates the need to iterate through all the vertices from start to end. This is perfectly visualised in the figure below.

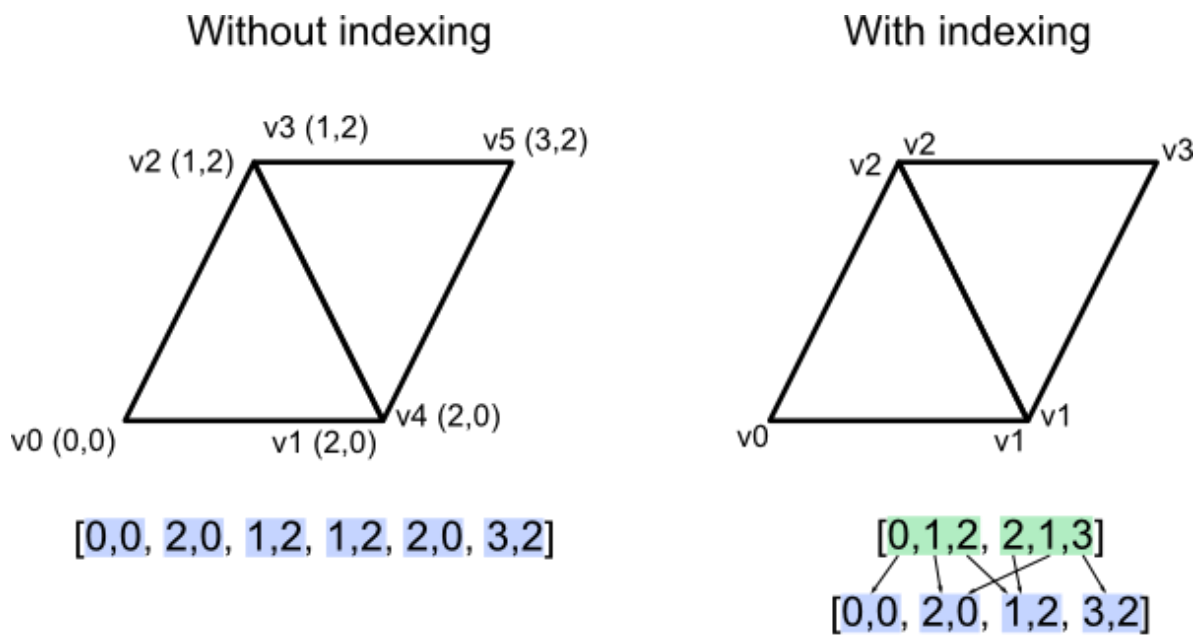


Figure 14: Difference between no indexing and indexing. [28]

Above is shown the differences between indexing and no indexing, let us explain.

Let us treat the above vertices as follows;

A – V0, B – v1 and v4, C – v2 and v3 and treat D as v5

Without indexing the program must specify all the vertices so in the above figure that would be, A, B, C, C, B, D. Hopefully you can already see that B and C are repeated. This is both a waste of space but with today's hardware it's more of a performance issue. Using indices addresses this issue.

With indices, you introduce a new buffer, named Index Buffer into the equation. The vertex buffer unlike before, now does not need to hold all the vertices, instead it would contain A, B, C and D, the Index Buffer would then be utilised and include the indexes of the vertex to draw.

Vertex Buffer: A, B, C, D

Index Buffer: 0, 1, 2, 2, 1, 3

The vertex data itself is not duplicated. Saving both space and improving overall performance.

Now that this has method has been introduced, it is used to store the geometry data needed. The storing / tile tracking system can now be addressed as previously indicated.

3.5.3 Tile system

Due to the requirement, the need for an infinite world. Generating the world when the program first loads would be impossible. Therefore, a way to split the world into different sections is needed, and a way to generate a new section is needed. These sections are generated around the player's current location.

This is thanks to the TerrainManager class. As suggested this class is used to determine if a new section of the world needs to be generated, if so tells the program to generate it. The class holds variables as to where terrain exists and the corresponding terrain. As the player moves around the current tiles tests are carried out as to whether new tile sections are needed. The system designed keeps a 5 by 5 grid of terrain around the players' centre tile. Therefore, as the player moves the centre tile is recalculated and the tests carried out again.

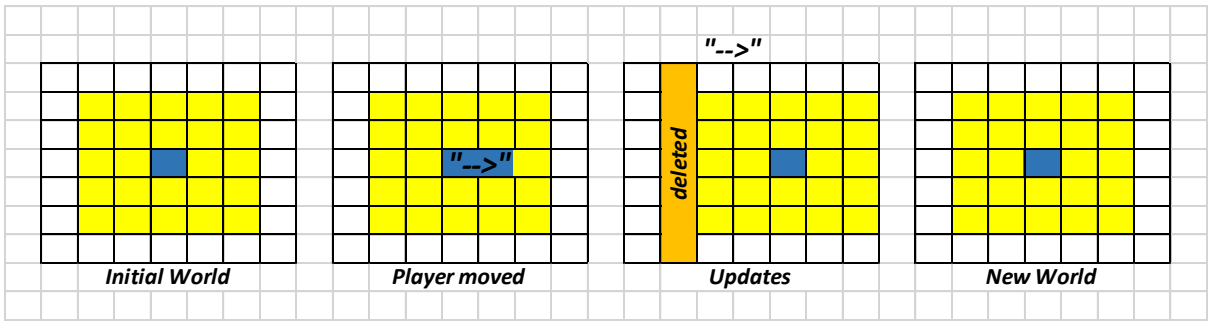


Figure 15: Mockup of how the terrain manager works.

Firstly, the TerrainManager class holds the current players' position and the location of each tile around the player. Whilst in the world each tile starts at (0, 0) locally and ends at (256, 256) in the terrain manager the tile position is required, that being that tile starting at (0, 0) and ending at (256, 256) is tile position (0, 0), then the tile starting at (256, 256) and ending at (512, 512) is tile position (1, 1). This works in the negative space too, with a little modification of subtracting one if the position is negative, so that instead of calculating the top corner we're calculating the bottom one.

Once the TerrainManager knows where the player is in relation to the grid system, tests are carried out on a 5 by 5 area around position.

These include, does a tile exist, does it need to. The class is designed to keep 25 tiles in memory at all times, a 5 by 5. Anything outside this area is deleted from memory, if during a test, it gets detected that a tile is needed but is not in memory, one is generated there and then, using the current algorithm selected by the user.

Due to the use of vectors as the storage method, adding and removing the tiles from memory is simple. Vectors have a function '.push_back' which adds the new data to the end of the vector, because of this and the way the system is coded new tile geometry data and the positional vector are updated after each other, imagine it as if it were done within the same call. A new terrain tile is not generated without the previous tile and position vectors being updated from the last 'newtile' call.

Therefore, index one of the tilePosition vector corresponds geometry data in index one of the tiles vector. When the system detects that a tile is no longer needed, we need to remove it from memory, otherwise with our world being infinite, we will eventually run out of memory, due to memory leaks. Memory leak is when a system is allocated memory but does not release it, eating up memory

available for other applications, eventually slowing or even crashing the application. Therefore, the unneeded geometry data must be removed. Due to the way in which we set up the two vectors, this is easy.

```
BEGIN
FOR 'EVERY FRAME' DO

      GET PLAYERS CURRENT POSITION
      BEGIN
      FOR '5X5 AROUND POSITION' DO

            BEGIN
            IF NOT CHECK TILE THEN

                  BEGIN
                  NEWTILE(POSITION)
                  END
            END
      END

      BEGIN
      FOR 'OUTSIDE 5X5 AROUND POSITION' DO

            BEGIN
            IF CHECK TILE THEN

                  BEGIN
                  DELETE(POSITION)
                  END
            END
      END

END
```

Code 1: Pseudocode for creating and deleting tiles on the fly.

Retrieve the index in one of the vectors of the tile needing to be removed. Remove it from the position vector, the index will match up to the correct geometry data, so delete, make it null (tiles vector holds pointers so the system must clean-up correctly, unlike tilePosition which just holds stack variables) and erase it from the tiles vector. The code for this system, is ran every frame or game loop. The player can move in any direction at any time, so a continuous check of what we need is required.

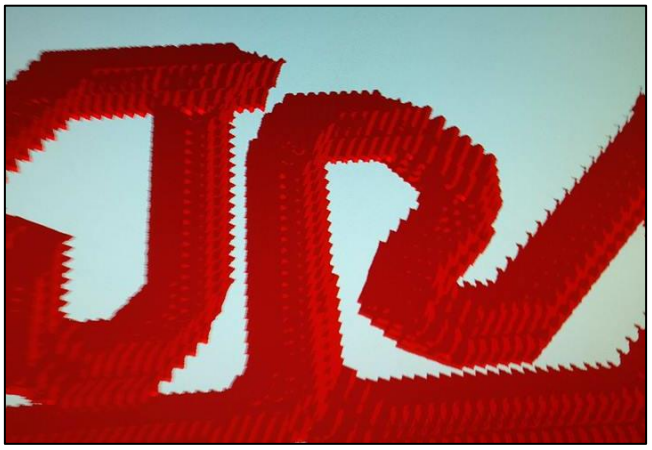


Figure 16: Showcase of the grid system, allowing tiles to be generated at any x, z coordinate

The figure above [Figure 16] showcases the very early stages of the TerrainManager, this version is full of memory leaks. As mentioned deletion of tiles that are no longer needed should occur, in this early version this was not the case. This was a test of the generation system, showcasing that new tiles could be 'spawned' / drawn at any given X, Z coordinate it was given. The input here was the player location, hence the messy placements. It was a proof of concept and it worked. The grid alignment and the use of tilePosition vector came afterwards. At this point the player could fly around an infinite world, terrain would 'appear' in front of them and 'disappear' behind them in all directions. Success. Requirement 1 and 2 complete.

3.5.4 Techniques.

This section will cover the techniques in greater detail, explaining how each one is used to generate the terrain.

One thing to remember is that even between the different techniques things remains the same. The size of each tile and therefore the amount of data that each tile represents is the same. 256 by 256. What the data is therefore, number of vertices and indices. Whilst the vertices themselves are going to different in terms of the Y values the amount of them is the same and it's how the Y value is generated that we care about.

3.5.4.1 Heightmap [29]

Heightmap was the first addressed. Due to NCLGL introduction series providing a tutorial about how to load files and extract the data from a RAW file alongside the indices and index buffer tutorial.

As has been previously touched on, Heightmaps can be used to store the Y values. These values can then be directly converted into terrain data and rendered. To achieve this first a Heightmap file was needed. This was straight forward. Included in the NCLGL framework was a RAW terrain file. It was generated using a program called TerraGen and all it is, is a file that contains 66,049 unsigned chars. That's 257 * 257. You can imagine that 257 * 257 as one of the tiles from the TerrainManager. That's 66,049 unique Y values we have access to. Just what is needed.

3.5.4.2 Diamond-Square Algorithm [30]

> *"perform a diamond step which sets the midpoint of the square based on the average of the four corner points plus a small random value, this is followed by a square step which sets the midpoint of the diamond to the averages of the four corners and another random value." [Online Midpoint Displacement generated terrain.]*

The second technique as stated is the Diamond-Square. As with the previous technique and implementation was researched. During the initial research into midpoint displacement algorithms a forum post regarding the diamond square algorithm came to life. It was regarding the 22nd Ludum Dare (Ludum Dare is a 48-hour game development competition).

Notch, the original developer behind Minecraft, had developed a 2D version. Unlike Minecraft which uses a Perlin Noise implementation or the level generation this uses a Diamond-Square function. This was the starting point. Further research into the area pointed towards a Java implementation derived from the Ludum Dare entry and written by Charles Randall. (See [31]) This find was key to properly understanding the algorithm in question. Each step is explained alongside with a visual representation

of the what happens. More research lead to another implementation this time in C++ and confirmed working by the author, Jimmy. [30]

The core of the algorithm was provided within the forum post: the implementation and setup was needed. From the previously found Java implementation it as was easy to get the setup working in C++ so as to utilise the C++ implementation mentioned.

Setup for diamond-square involves setting the initial size of the tile. A tile in this instance being the variable which holds the values generated in the core algorithm. This was chosen to be a 2D array size of 257 by 257, giving a working array size of 256 by 256. That's the same size as the tile that's going to be drawn. In this setup, each tile therefore terrain section is generated by its own call to the diamond square function. Afterwards to extract the generated value, because the sizes match, it's a case of simple indexing correctly.

```
float noise = dS.getValue(i, j);
noise = abs(noise);

y_values[(i * RAW_WIDTH) + j] = noise
```

Code 2: Mapping from an 2D environment to noise values.

3.5.4.3 Value Noise Function [15]

During initial research into procedural generation and noise functions, if you recall the site scrathapixel was referenced. This site provided insight into what a noise function, but unaddressed previously it also provides a C++ implementation of Value Noise, more specifically a 2D implementation, which is perfect. This means that just like the Diamond Square algorithm the noise function will take two inputs, an X and an Z.

Like the diamond square function some setup work is required before the noise function can be called. This setup involves first setting initial values, but unlike the diamond square this step is pre-populating the entire data structure (again an array) with random values. This is known as a permutation table and it is this table that is used for the interpolation between the mapped values mentioned next.

Unlike the diamond square technique however, the 2 input parameters are of type float not int. This means that the X and Z coordinates, players position, can be a decimal place, .34 or .87 for example, whereas the diamond square only offer whole numbers as the position inputs due the grid system mentioned above. You may be able to see that due to the use of decimal places our get value function might be a value in between values that have already been mapped see Figure 17.

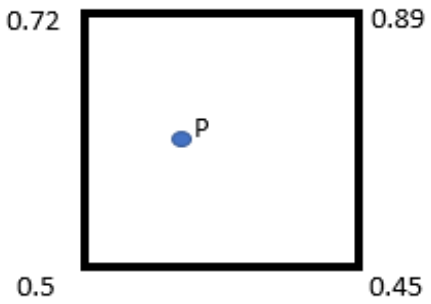


Figure 17: Chosen point P may lay between mapped values

This is where the value noise gets the name. So that Value Noise can get value at this position the algorithm uses interpolation between the current mapped values. Value Noise.

Interpolation between the mapped values is carried out on both the X and Y axis.

First linear interpolation along the bottom X-axis is done, followed by the same on the top X axis. This results in two new values, which can then be used to carry out interpolation along the Y axis. This process is otherwise known as bilinear interpolation. This process results in a value at P being returned. This new P value is the noise value and this is the Y value used for the terrain geometry generated from Value Noise.

3.5.4.4 Perlin Noise Function

Research into Perlin Noise was relatively straight forward. This noise function is very well documented mainly in part for the fact it is widely used and was the first noise function that uses a gradient value technique not like value noise mentioned above. Once again, the same as with Diamond Square and Value Noise, during research into the technique itself a C++ implementation was found. That being an implementation written by Luke Benstead and posted on blog.kazade.co.uk under the unlicensed format.

> *"Recently I needed to generate some simple procedural textures for a game I'm working on - after a quick search I found that the top result was licensed under the GPL which is no good to me. …, but as all these code snippets are just C++ ports of Ken Perlin's improved noise function written in Java, I figured I might as well write my own from scratch ..."* [32]

The implementation found and used is a 3D version of the noise function. Whilst this is different to the implementation used for Value Noise this is acceptable as the other gradient noise technique Simplex is also of a 3D environment.

Just as with Value Noise the first step is to do some setup work this is similar but can differ slightly depending on the approach taken for the implementation. For this dissertation and the implementation used just as in Value Noise a table known as the Permutation Table is setup. This is the grid definition step. Other setups may change here and assign each grid point a random gradient vector of length 1, however this implementation the gradients are not stored but calculated when needed.

Once the gradient vectors are known the algorithm is very close to values noise. Due to the input parameters, the point we are searching for may not be mapped to anything, that being that it lays between our grid cells. This is where the gradients whether pre-calculated or not come into play. First the distance vector from the point we want and the surrounding cell points is calculated. The dot product of the distance and gradient vector is then evaluated. Just as in Value Noise there is an interpolation step that happens last. X axis first then along the Y axis. This gives point P a value, this value is the Y value for the terrain at the X, Z coordinate inputted.

3.5.4.5 Simplex Noise Function

Simplex Noise was the first noise function that was researched due to, it being known as the best function to generate noise. It claims this title because Ken Perlin published the algorithm so that he could address drawbacks in his original Perlin Noise function.

> *"In 2001, Ken Perlin presented "simplex noise", a replacement for his classic noise algorithm" "but in hindsight it has quite a few limitations. Ken Perlin himself designed simplex noise specifically to overcome those limitations, and he spent a lot of good thinking on it. Therefore, it is a better idea than his original algorithm."* [33]

The abstract above comes from a paper by Stefan Gustavson, titled Simplex Noise demystified which explains the benefits Simplex has over Perlin and importantly how it works.

Simplex Noise offers plenty of advantages over Perlin Noise;

- *it offers lower computational complexity, fewer multiplications.*
- *The big O notation is $O(n^2)$ instead of $O(2^n)$, no exponent N.*
- *No noticeable directional artifacts.* [16]

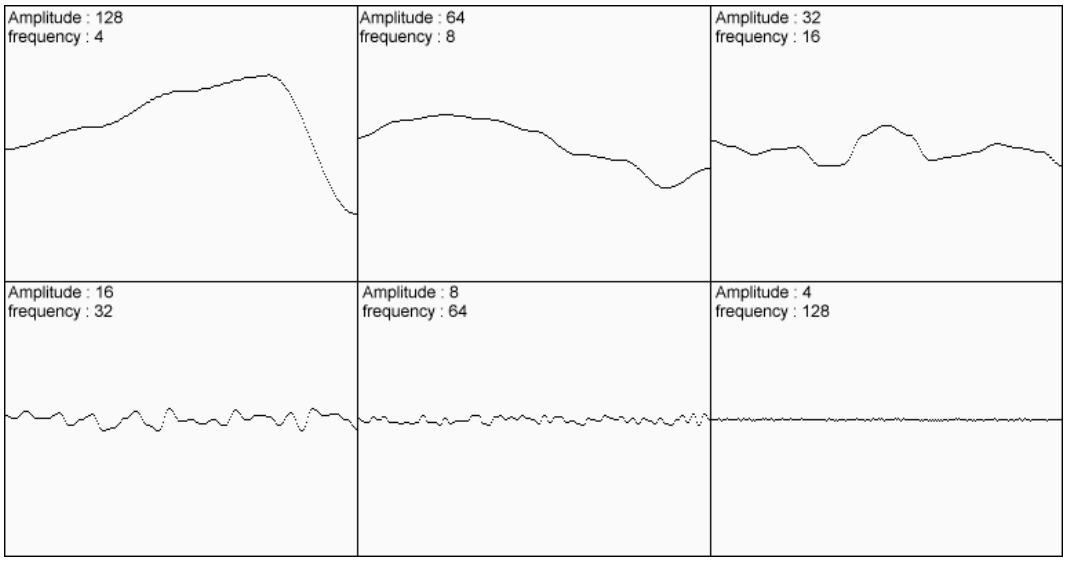
The biggest advantage being that it requires lower computational complexity and the big O containing no exponent property.

> *"In higher dimensions, even finding the noise for a single point requires a lot of math. Even in 3-D, there are eight surrounding points: eight dot products, seven linear interpolations, three rounds of smooth step. Simplex noise is a variation that uses triangles rather than squares. In 3-D, that's a tetrahedron, which is only four points rather than eight. In 4-D it's five points rather than sixteen."* [34]

However, whilst this is all good, it's important to note that for 3D and above implementations of Simplex Noise it is patented for certain situations/applications, whilst Perlin Noise is not. [35] This is further discussed during testing.

3.5.5 Octaves

Octaves is the name given to the process of adding the results of multiple noise functions at different frequencies and amplitudes together, in order to create a more natural-looking noise wave. The results of which can be seen below. This feature is being used within the tool developed for every noise function so that's Value, Perlin and Simplex Noise. The Diamond Square and Heightmap techniques do not use octaves function.



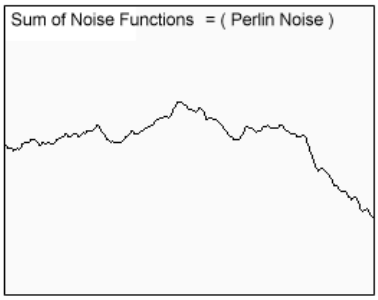


Figure 18: How adding different level of noise works to create an overall noise wave. [36]

3.6 Validation

This section will briefly discuss the way in which the techniques were confirmed and implemented correctly before testing was properly carried out.

3.6.1 Heightmap

Due to the Heightmap generated terrain representing values within a text file, getting confirmation that this implementation was correct was straight forward. Compare the values within the program to the text file outside the program. This was done using Notepad++ and the compare plugin.

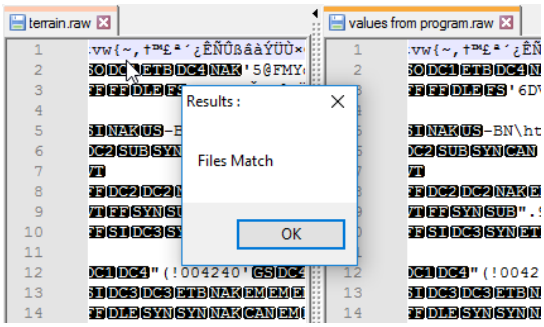


Figure 19: Showing that the original and new files match.

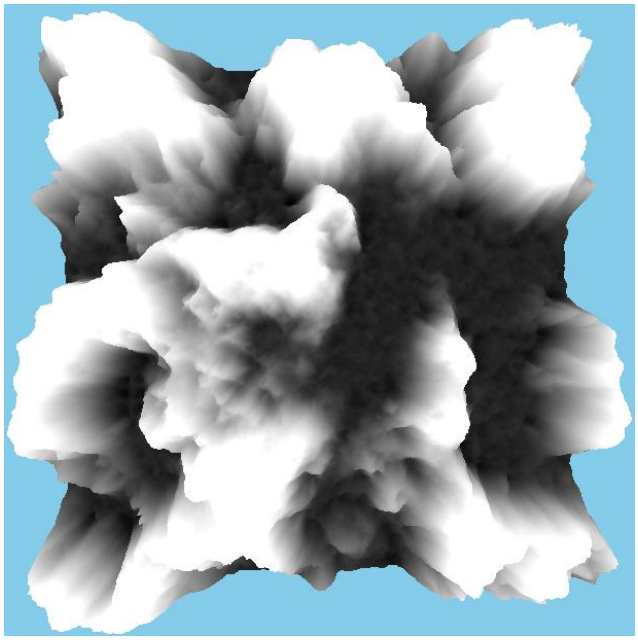


Figure 20: Heightmap from the dissertation tool.

The above cannot be used for the other techniques because of these noise functions, so that we can to see if the tool was acting as expected a simple visual comparison between the tool output and examples of that technique were carried out.

3.6.2 Diamond Square

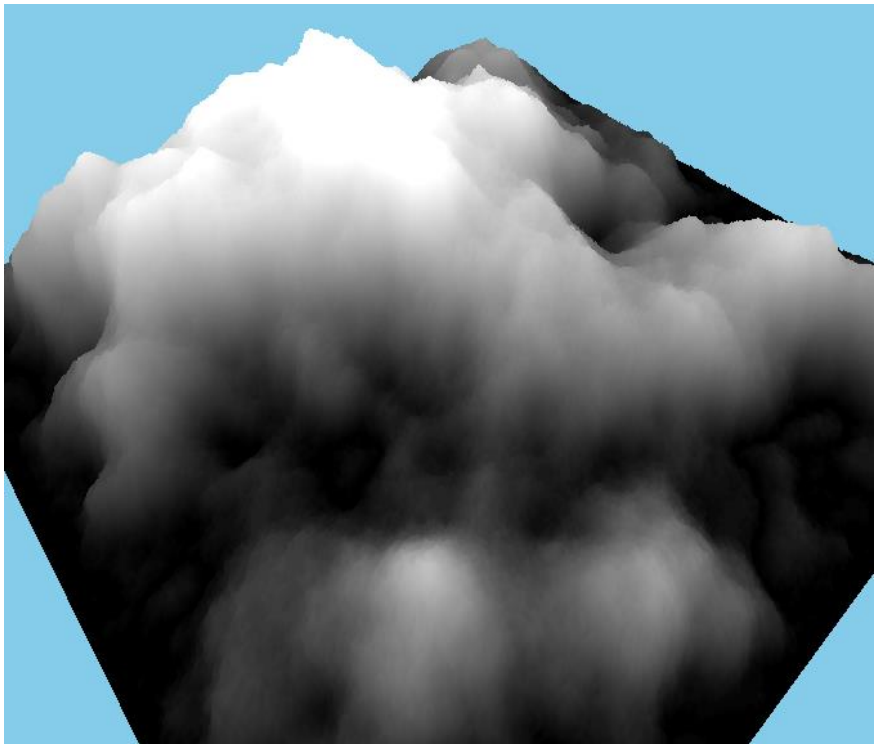


Figure 21: Diamond Square from the dissertation tool.

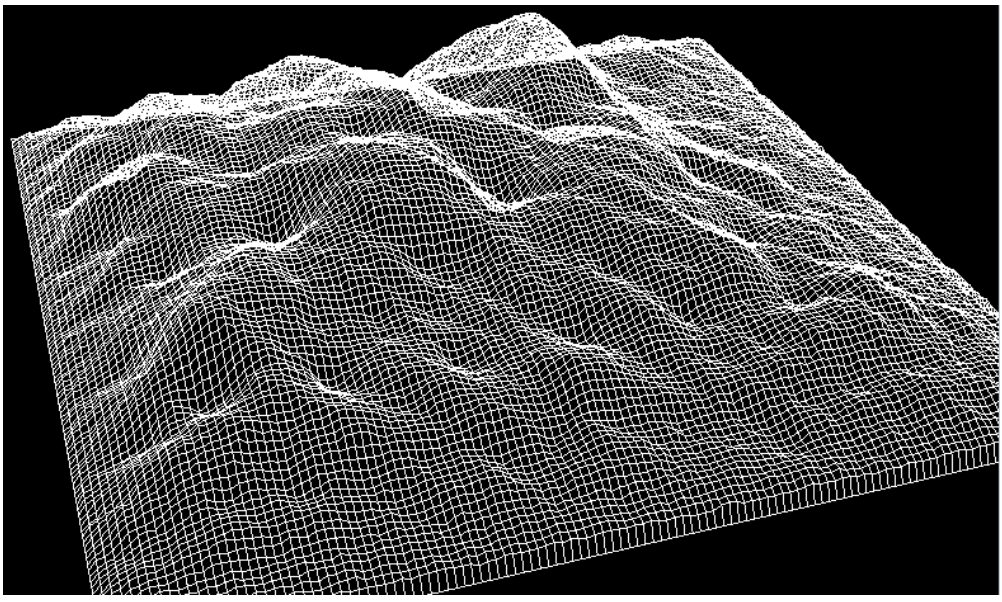


Figure 22: Sample by the author of the implementation used. [30]

The trend with Diamond Square is for the edges to be small values and work upwards as the algorithm moves in. This is the tools results, and the result of what the authors implementation achieved.

3.6.3 Value and Perlin Noise

Value Noise as touched on briefly in section [2.2.3 Noise Generation] and visualised in Figure 8: Value vs Perlin 1D output. , it's shown that value noise trends towards smoother looking noise with less leaps between values. This is what the tool outputs, smooth looking noise. Gradual change in values.

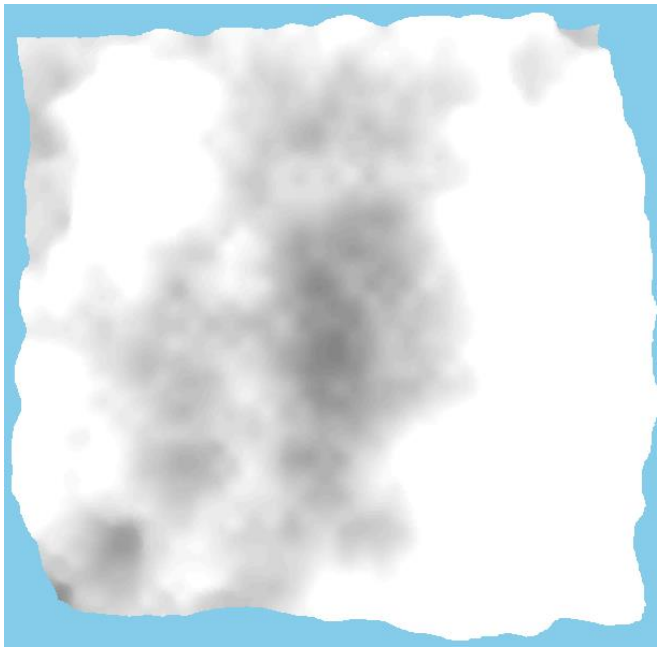


Figure 23: Value Noise from the dissertation tool.

Perlin Noise however as address in the same section as above, has a less smooth looking output. It can show large leaps in values. Faster change in values.

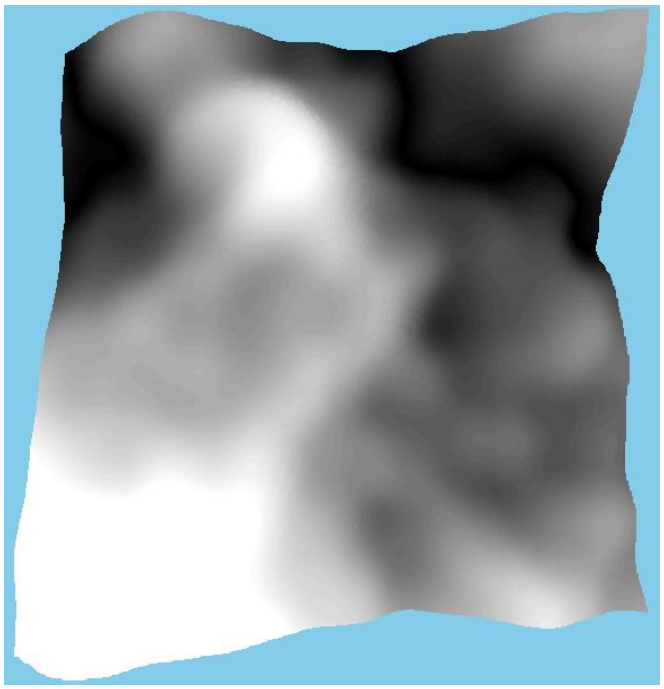


Figure 24: Perlin Noise from the dissertation tool.

3.6.4 Simplex Noise

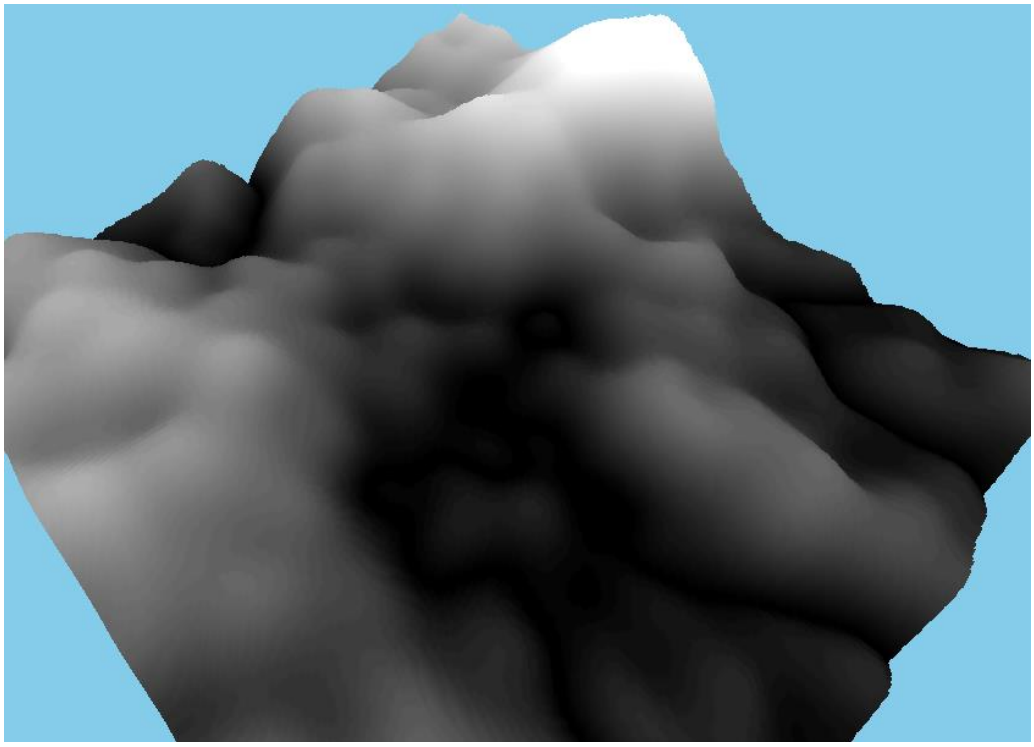


Figure 25: Simplex Noise from the dissertation tool.

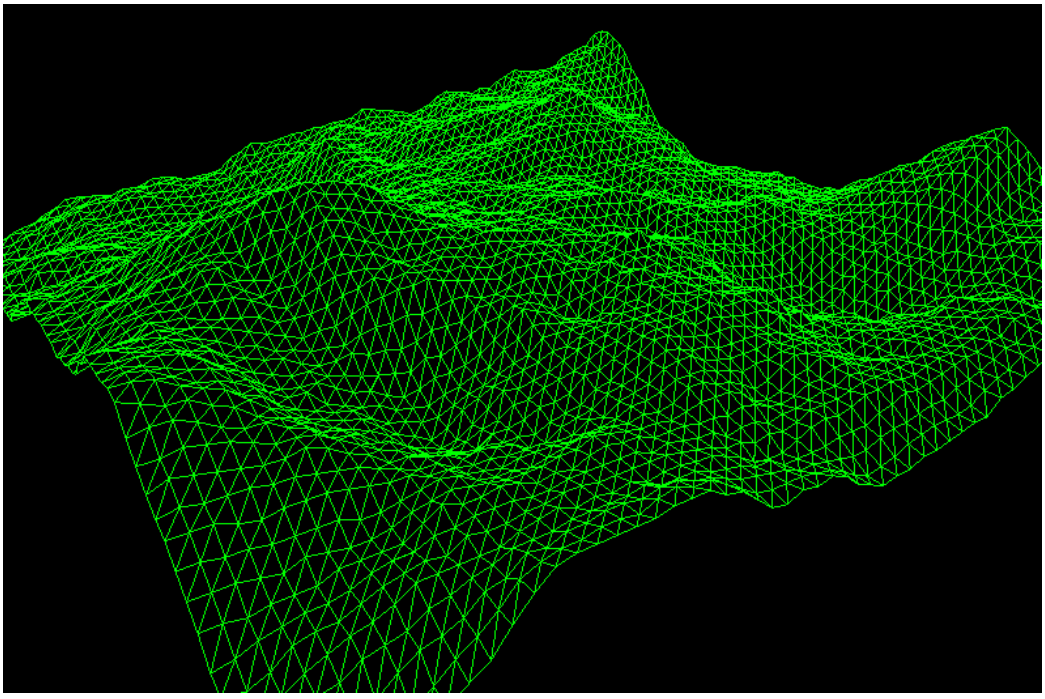


Figure 26: Terrain generation from Simplex Noise. [37]

As discussed Simplex is based on Perlin, and improves upon it in regards to the higher dimensions. As this dissertation focuses on 3D implementation. It is not unexpected for the output to be similar. Also, our output lines up with the terrain generation by a forum user from another Simplex Noise implementation.

3.7 Summary

This section has gone over the development that was undertaken so that we could have a tool that could be used to evaluate the different techniques that could be used in creating procedurally generated terrain in real-time simulations.

It introduced the methodology for this project, covered the tools that would be used for the development portion and the final set of that the tool implements was decided upon.

The chosen development language and the tools such as the IDE and any APIs that were used was introduced and a brief summary given. The projects requirements were covered and explained as to how they were measured.

Development. The development was covered, in which the concept and theory behind the use of indices was covered and the tools tiling system that was developed to meet requirement 2 of the playable world being infinite was addressed. Following each technique was covered independently as to how the underlying algorithm works.

Finally, the method of how each technique was verified to be correct was covered and shown as to why the results are valid.

Chapter 4 – Evaluation

4.1 Overview

This section discusses the findings from this dissertation project, providing results in the form of graphs and figures of the terrain generated from the tool developed for this project. Findings such as the FPS, RAM usage and GPU usage were looked into.

4.2 Testing Solution

First thing for testing was to ensure that each time a test was ran, the underlying framework was the same, as to only having the algorithm that generated the values to render as terrain to be different. As mentioned this was accounted for at every stage during development. By having the testing in mind for the development phase of the framework it is true to say that only the algorithm changes technique to technique. The titling system developed ensures the same amount of geometry is kept active throughout, this also means that the amount of RAM being used remains the constant.

The second part that being only the algorithm changes means the user inputs must be the same. Ensuring that the user moves in the same way, direction, speed any pauses taken are for the same length with the end goal of always ending the benchmark in the same X, Z coordinate.

4.2.1 Auto Hotkey

Knowing that the framework in which the techniques were implemented into was consistent, and didn't change between them, a method that would ensure the user inputs were consistent needed to be developed. AutoHotKey was the answer.

> *"AutoHotkey is a free, open-source scripting language for Windows that allows users to easily create small to complex scripts for all kinds of tasks such as: form fillers, auto-clicking, macros, etc."* [38]

With AutoHotkey the user inputs can be scripted, meaning that in order to carry out a test only 1 input from outside the OS would be required, then the script would take over ensuring that the terrain being generated is always the same in regards to its X and Z coordinates.

4.2.1.1 Testing Script

It was decided that due to the nature of this dissertation being infinite world, the test script that would be needed was a simple one. Press and hold the move forward key for a period of time, move the camera angle a bit during this time and press the move sideways key for a period of time. Whilst AHK does support very complex macros and the like, the need in this dissertation is simple. The script in question is visible in Appendix A.

In all, what the script does is when the 'p' button is pressed, it sends a command to the OS that tells it that the 'w' button was pressed down. Notice down only. This means that the OS now thinks that the w key is being held down, and processes as such, in terms of the tool this translates into player movement in the camera direction. After 300,000 milliseconds (5 minutes) the up command for 'w' is sent. This is followed by a 'w' down and 'a' down for a further 5 minutes, after which an up command for both is sent.

Whilst this script is simple it does what is needed to be done. The user is simulated in such a way that the tool is subjected to consistent inputs and requests for the noise value at the same X and Z coordinates. Perfect.

4.2.2 FRAPS (cont.)

Fraps has been mentioned already in this dissertation however, it needs to be addressed again in terms of how it was utilised for the testing. As touched on Fraps is a windows benchmarking tool, it displays the FPS of a DirectX or OpenGL windows application, it also includes a couple of very nice features.

These include;

- The ability to do a timed benchmark.
- The automatic calculation of minimum, maximum and the average FPS.
- The inclusion of a hotkey to start the benchmarking.

Therefore, FRAPS was used in combination with the script, because it has the option to assign a hotkey which means starting the Fraps benchmark and the AutoHotKey script could be assigned to the same key and they would both start together and end after the same amount of time.

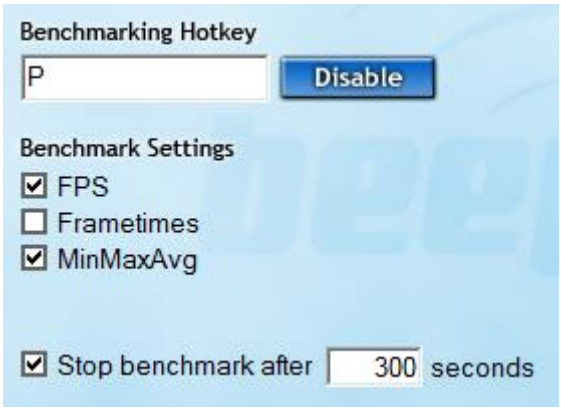


Figure 27: Screenshot of FRAPS setup.

4.3 Results

4.3.1 FPS

As stated FRAPS was used to measure the FPS of the tool developed. More importantly focusing on the difference between each technique.

Down to the way that the framework was developed that being no threading taking place, and the renderer running after the techniques have generated the values means that at times the tool from a user's point of view appears to freeze as nothing new rendered on screen and player movement is blocked. However, this can be used as a testing situation.

During the time that a technique is generating new values, i.e. when the player is moving to a new centre tile, the fps achieved will be less, sometimes as low as 7fps. (in release mode).

Let's first look at the minimum fps to begin with. Heightmap technique is a clear winner in this category with the next nearest 66 frames down, from these results alone its clear to see also that Perlin Noise is dragging behind any other with the lowest value at 7fps, the nearest to it is Simplex Noise but with much more respectable 28fps.

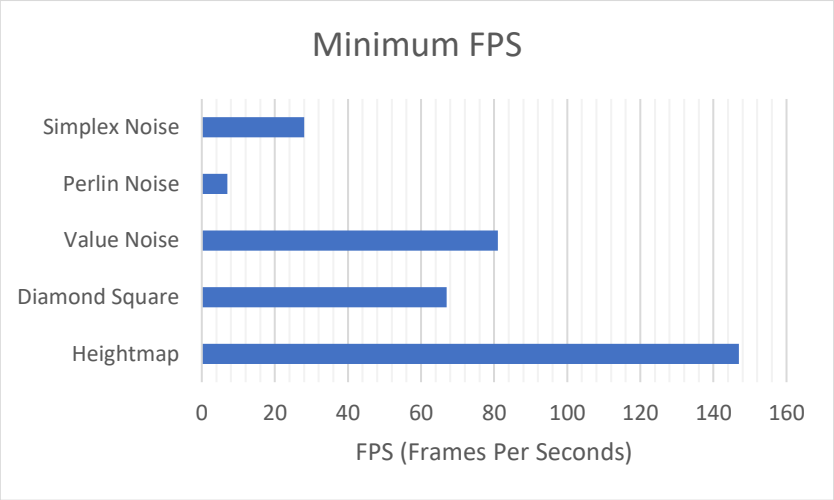


Figure 28: Minimum FPS achieved from each technique.

Not much can be said regarding the maximum values plus or minus an error value, depending on what the hardware was otherwise up to and the results are equal. 206/7 across the board.

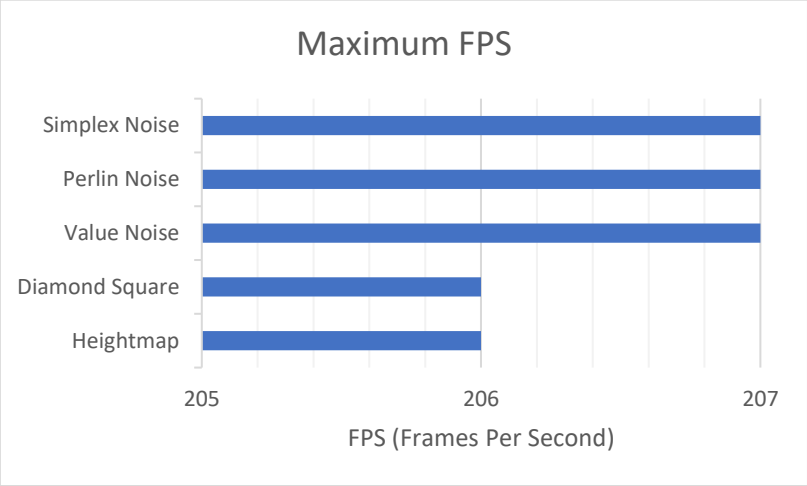


Figure 29: Maximum FPS achieved from each technique.

If the tests stopped here and based on solely the min and max fps then the Heightmap would win outright, Value noise would be the closest with diamond square close third followed by an acceptable yet underwhelming Simplex noise and Perlin noise bringing it home.

However, the above does not tell the complete story. The average fps has to be taken into account. It’s one thing to say that the max fps is high but if the average is low then it is irrelevant.

There is a big debate within the video game industry into the fps of games and what it should be. Should it be 30, 60, 120fps, stable, never drop below that value, is the issue is that the game industry has no standard. It is a very strange environment. In terms of this dissertation a higher average fps such as anything above 60fps will be deemed a good implementation.

Speaking of which.

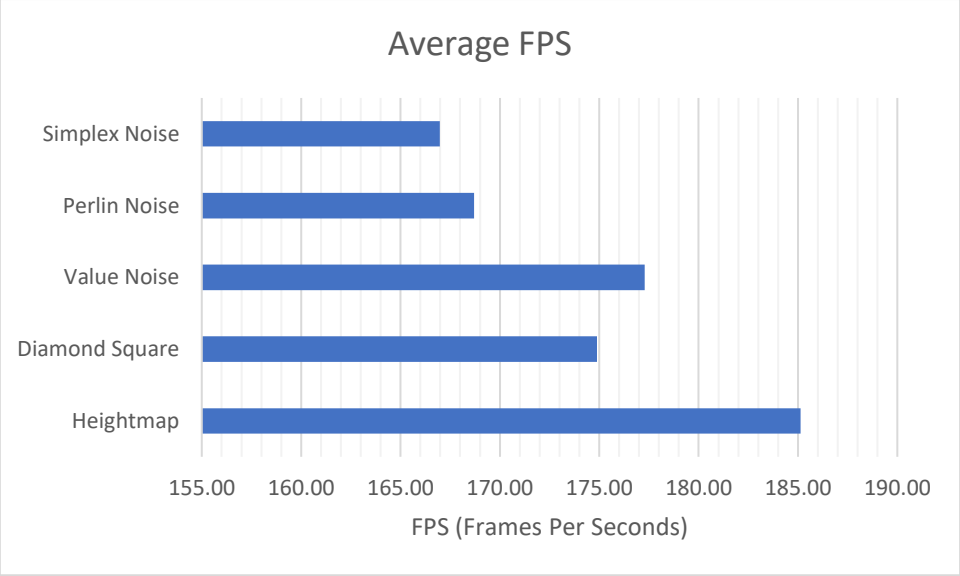


Figure 30: Average FPS achieved from each technique.

These results are more comparable from one another therefore, let focus on them. As before Heightmap is once again pulling ahead averaging 185fps over the full 10 minutes, but with the others very much closer than when comparing the minimum values before. The other techniques par Simplex and Perlin are in the same order as the minimum. Simplex and Perlin are reversed, but within very close proximity at 167fps and 169fps rounded up respectively.

Shown is that each technique has a high frame rate as it easily surpasses the mark of 60fps average being deemed a good implementation. All those covered thus far are good in terms achieving the best fps however, other factors do come into play when looking at the entire picture.

Algorithm Name	FPS			
	Min	Max	Average	Difference
Heightmap	147	206	185.12	59
Diamond Square	67	206	174.88	139
Value Noise	81	207	177.29	126
Perlin Noise	7	207	168.70	200
Simplex Noise	28	207	166.98	179

Table 1: FPS data from FRAPS.

4.3.2 RAM

Due to the underlying implementation, the RAM used should not vary technique to technique. This is known because the tiling system created has been created to always keep 25 tiles in memory, with each tile being the size data wise.

```
/*iterate through the vector removing any tiles and tiles position
    information we no longer need, remembering that they align with eachother. */
    for ( auto it = tilePosition.begin(); it != tilePosition.end(); ) {

         // if it lies outside our 5x5 area nuke it
        if (it->x < (x_ - 2) || it->x >= (x_ + 3) || it->y < (z_ - 2) || it->y >= (z_ + 3)) {

                ...
        }
    }
```

Code 3: Deleting out of bounds tiles.

What happens is every frame, checks are carried out to see if a new tile is needed, if so, one is generated as mentioned previously, when this happens then tiles outside the 5 by 5 area need to be removed, the code snippet above handles this check.

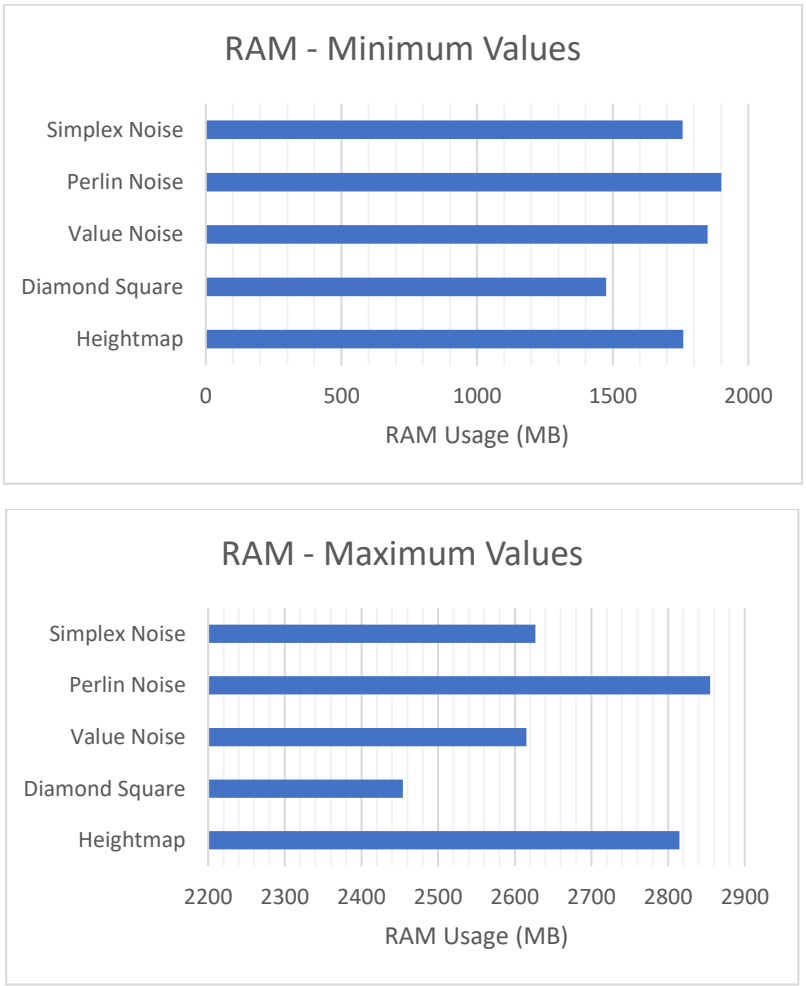


Figure 31: Maximum & minimum RAM values from the techniques.

The two graphs above focus on the RAM values achieved during the tests, the same tests as carried out for the FPS testing, so that is 10 minutes of the tool running. Now initially these results do look a little strange, throughout this dissertation it has been reiterated countless times about how the underlying framework was designed to keep everything the same apart from the algorithm or technique that is being used. So surely if this was the case the results would reflect this. Unfortunately, it is not as straight forward as that.

First, it's important to remember that background apps and services are always running on a computer, as such these may have an impact of the hardware, in this case RAM usage.

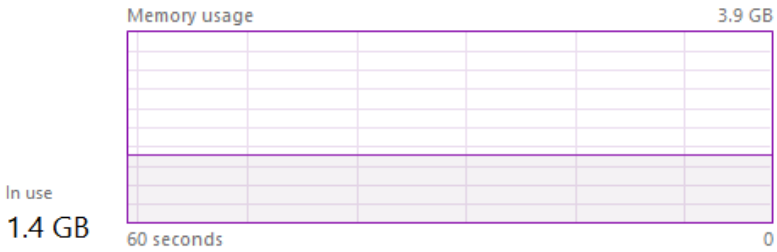
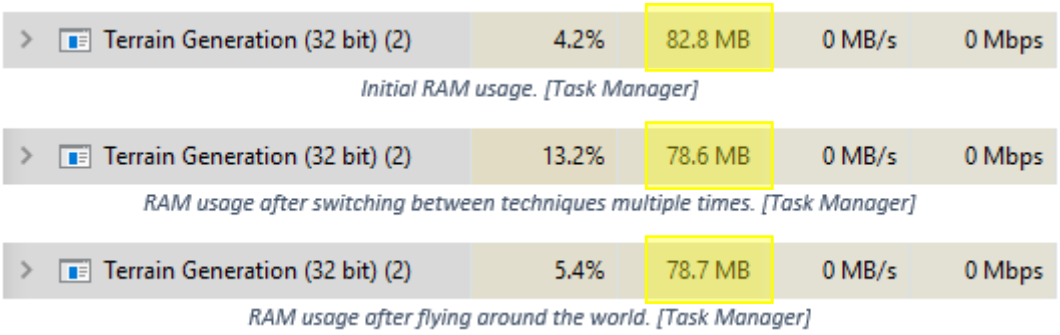


Figure 32: Base RAM usage on start-up

The figure above is a capture of the RAM usage over the last 60 seconds, 60 seconds after the machine had been restarted. This shows that with nothing running (the user hasn't started to use the machine) the RAM usage is at 1.4GB or 1400MB, it's also important to remember that the background services or apps that are running can create or destroy others as to fluctuate the RAM usage. Therefore, the minimum and maximum values shown in Figure 31: Maximum & minimum RAM values from the techniques. aren't the true values for the tool itself.



Initial RAM usage. [Task Manager]

RAM usage after switching between techniques multiple times. [Task Manager]

RAM usage after flying around the world. [Task Manager]

Figure 33: RAM usage at different times.

The three above figures show the true RAM usage of the developed tool at various stages that include, at rest, after switching countless times between then techniques and after the user has moved positions multiple times, as indicated by the windows task manager. These readings give a far better example of the framework, properly creating and destroying the tiles throughout the programs execution. Keeping the RAM usage of the tool similar technique to technique. Only the technique changes as stated in requirement 5 of this dissertation project.

4.3.3 GPU

One final piece of the hardware to look at is the GPU (Graphics Processing Uni). This has a direct impact on the FPS discussed above. The issue with discussing the GPU usage is that the usage is not affected by the technique. The same number of vertices are being rendered by the GPU every time. One frame does not include less vertices than the next, thanks to the framework and tiling system. Another issue with the GPU is that Nvidia has developed a technology called GPU Boost 2.0. Which is designed to vary the clock speed of the GPU depending on factors such as GPU temperature in order to maximise the GPU to its full potential.

Meaning the GPU usage graph for the tool looks as follows.

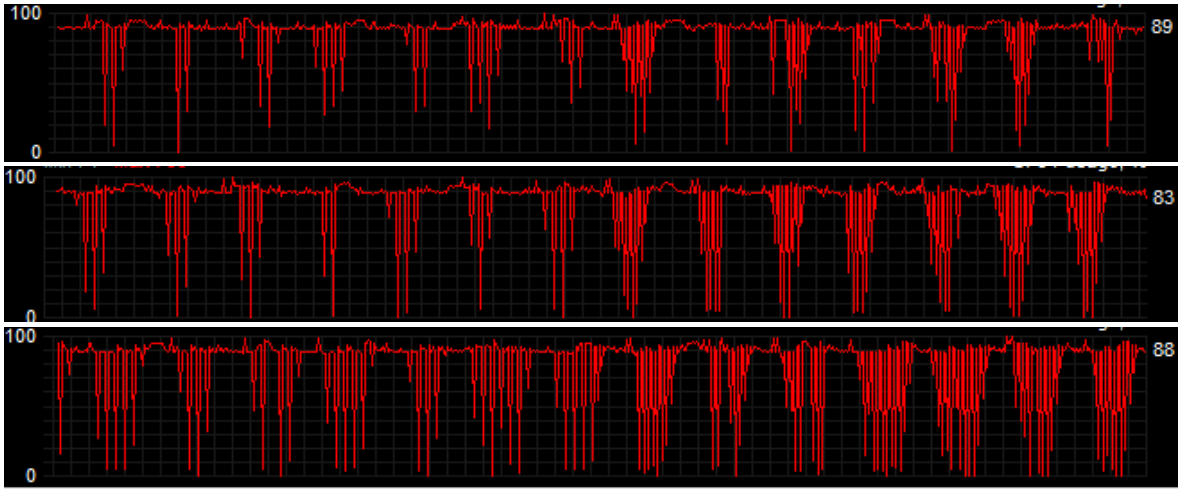


Figure 34: GPU usage graph whilst tool is running.

Unlike previously where noise was good this noise is not nice. This is from the GPU Boost mentioned above and the impact thermal throttling has on GPU Boost. Testing was carried out using a laptop therefore cooling is an issue. As the work load rises so does the temperature, laptops are not the best at dealing with this and therefore the GPU clock gets throttled/limited. The MSI Afterburner and other monitoring software have a difficult job of accurately reading the correct load or usage when the clock rates fluctuate, because of this the above graph is misleading. They show the GPU as having varying workload to carry out, which is incorrect. What is happening is as the GPU temperature rises the clock is tuned down whilst the workload remains the same therefore it seems a higher percentage is calculated, with the clock speed down the temperature can decrease so the clock can be boosted which, with the same workload shows up as less % usage this is repeated throughout the testing, therefore the conclusion from the GPU is that minus the external conditions such as air temperature which would impact the rate of which the GPU cooled, GPU usage is constant.

4.3.4 Generated Terrain

This section is going to focus on arguably the most important aspect of this project, that is what type of terrain is actually generated from each technique.

First off, to qualify for requirement 4, the terrain colouring needs to be addressed, in order to understand the differences between the terrains in the figures below. The system colours the terrain based on the Y values of the vertex that is currently being drawn. The code snippet from the shader

to do so is in appendix C, but briefly what happens is the colour of the vertex is being interpolated between fully blue and fully green. The higher the Y value the greener the resulting colour and vice versa for lower the Y values.

4.3.4.1 Heightmap

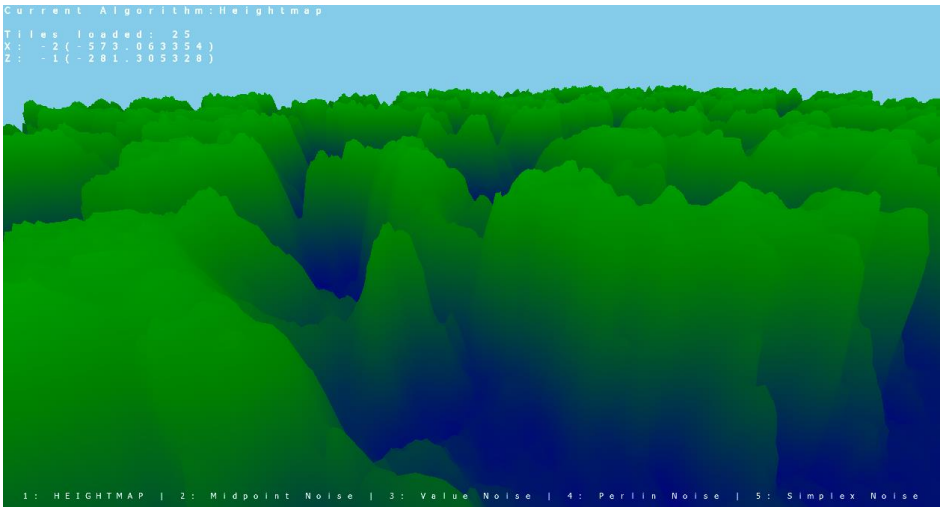


Figure 35: Terrain generated from Heightmap.

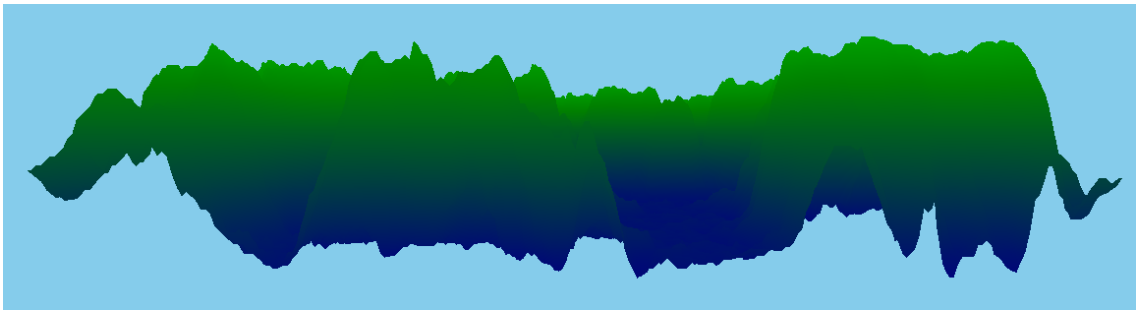


Figure 36: Orthographic view of a Heightmap tile.

The first as has been throughout is the Heightmap technique. Whilst initially the figure above shows terrain that stretches off the screen, it's not all as it seems as shown in the figure below.

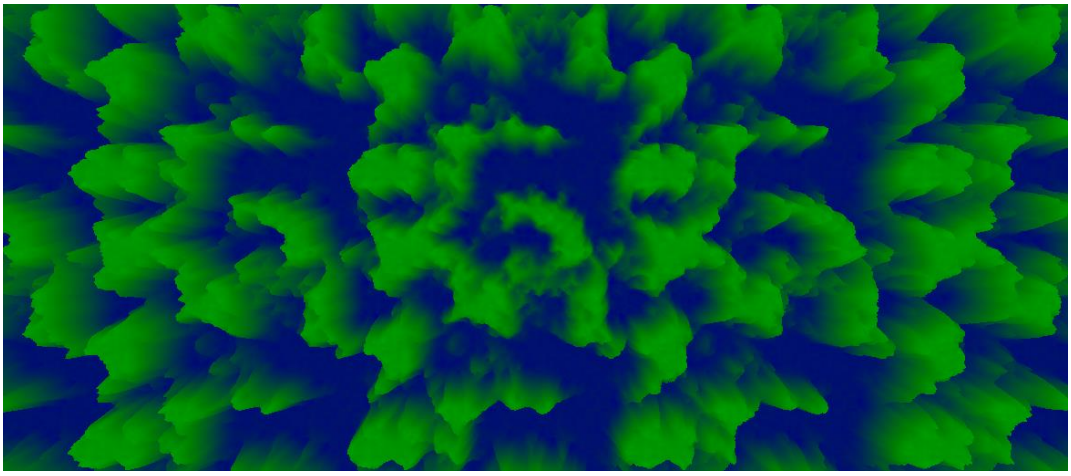


Figure 37: Heightmap repetition is very obvious.

Due to the nature of using a Heightmap the terrain is going to be all the same. Each tile is loading the same file data and therefore all the Y vertices will be the same with respect to the tiles. That is that X 128, Z 128 in tile 0, will have the same Y value as X 128 and Z 128 in tile 1, 5, 8, -45, 19, -5, -62 and on for all tiles that are in the system.

4.3.4.2 Diamond Square

The second technique is without a doubt produces a visually better-looking terrain than heightmap. Each tile is different and the overall terrain that is generated looks like terrain and not like a generic placeholder terrain file. However not all is as it seems. As stated in [2.2.3 Noise Generation] a noise function is pseudo-random that being that to the user the outcome seems random but in reality, the function is predictable. Diamond square is not this. Another drawback of the algorithm is that one tile has no influence on any of the adjoining tiles, this means the terrain whilst each tile looks at the least natural they are not over the border between tiles. Therefore, in the quest for an infinite world using the Diamond Square technique would mean that the terrain is always different and not because the user wanted it to be. As is shown in the figures below.

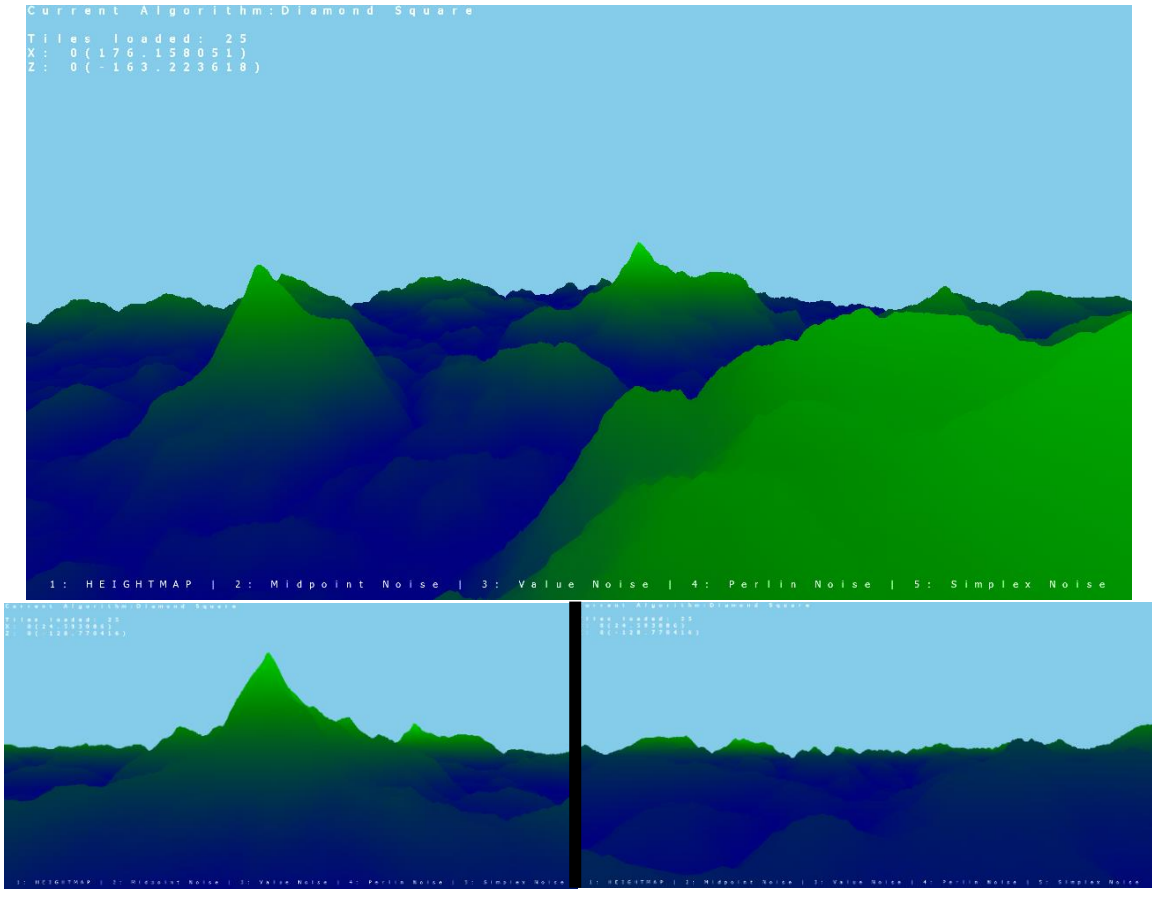


Figure 38: Terrain generated from using Diamond-Square.

As shown the terrain captured is from the same world position, tile 0, 0 and yet the terrain generated each time is vastly different. Probably not ideal for terrain generation, randomness is good but it must be controllable randomness, such as when using a seed value.

Moving onto the noise functions, it's known that due to the nature of noise functions these techniques also produces terrain that is repeatable, not repeatable as in the terrain itself as with the Heightmap, but as in an X, Z position will always produce the same Y value. Every time. Without question.

4.3.4.3 Value Noise

Value noise as addressed is the first noise function technique. Therefore, this technique is already a better technique in terms of visual terrain generated when compared with the Heightmap technique, the terrain generated just like Diamond Square is visually appealing. It does look natural, and because of the consistency between the mapping of X, Z to a Y value, value noise does not have an issue with borders between tiles.

The figure below also shows off the small changes in the Y values for changes in the X and Z axis. This technique produces long smooth slopes across the terrain, as is mentioned in Figure 8.

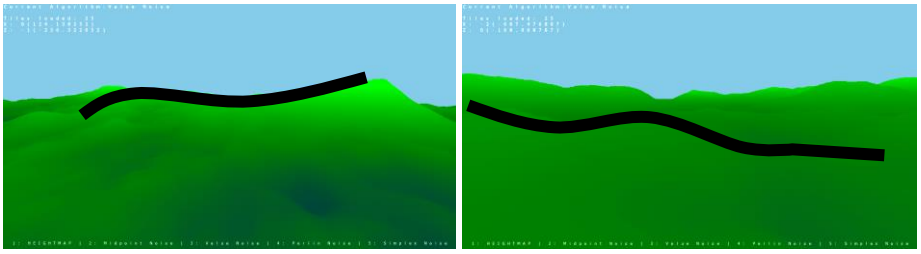


Figure 39: Smooth slopes mentioned in Figure 8 visualised.

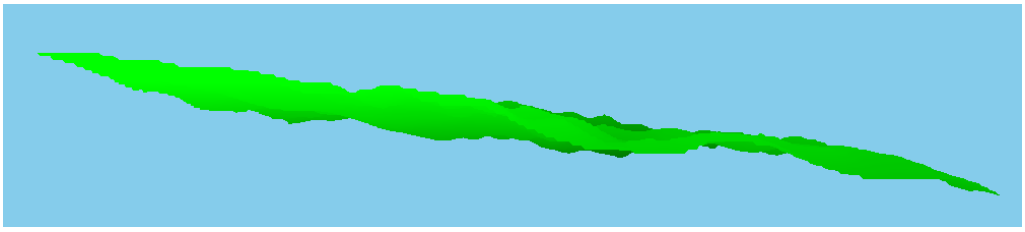


Figure 40: Orthographic view of Value Noise.

However, due to the smooth transitions between output values, whilst the terrain does look natural, it will always be a high terrain, this means that the minimum value is going to be high when compared to the other techniques. This is visible by the lack of the deep blue colour in the figures above and below.

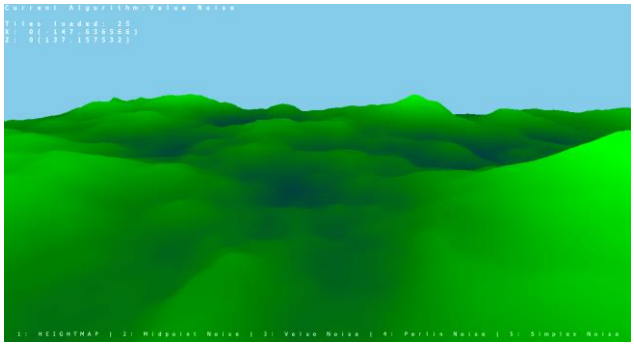


Figure 41: Terrain generated from using Value Noise.

4.3.4.4 Perlin Noise

The difference between Perlin Noise and Value is noticeably just from the terrain that it generates. Perlin does not suffer from the smooth changes in output values as value does, as addressed above, Figure 8 because of this the terrain looks even more natural.

The range of values that get returned from the noise function is greater therefore the variations in the terrain is greater. This is shown in the form of the terrain colouring and a more present blue value, which represents low values. So, whilst Value noise generates terrain that has gradual changes across a large area, Perlin generates sudden changes across a shorter area.

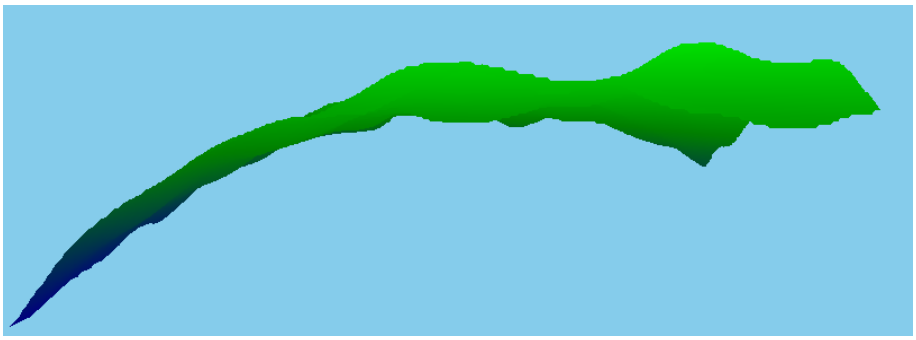


Figure 42: Orthographic view of Perlin Noise

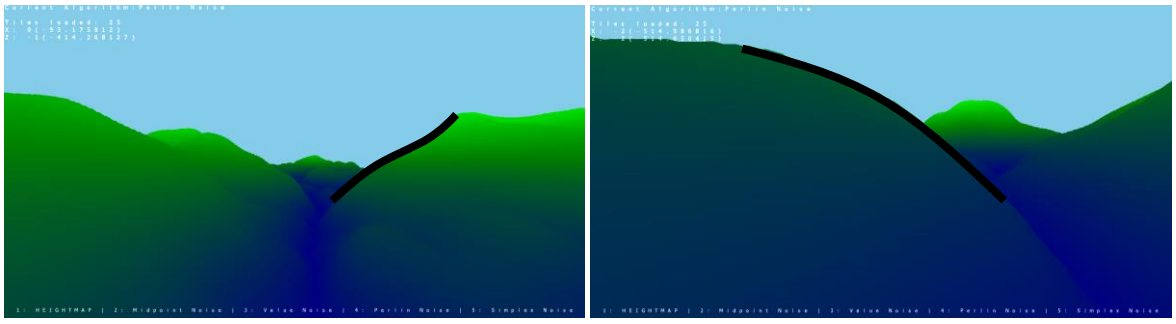


Figure 43: Lack of smooth output values compared to Value

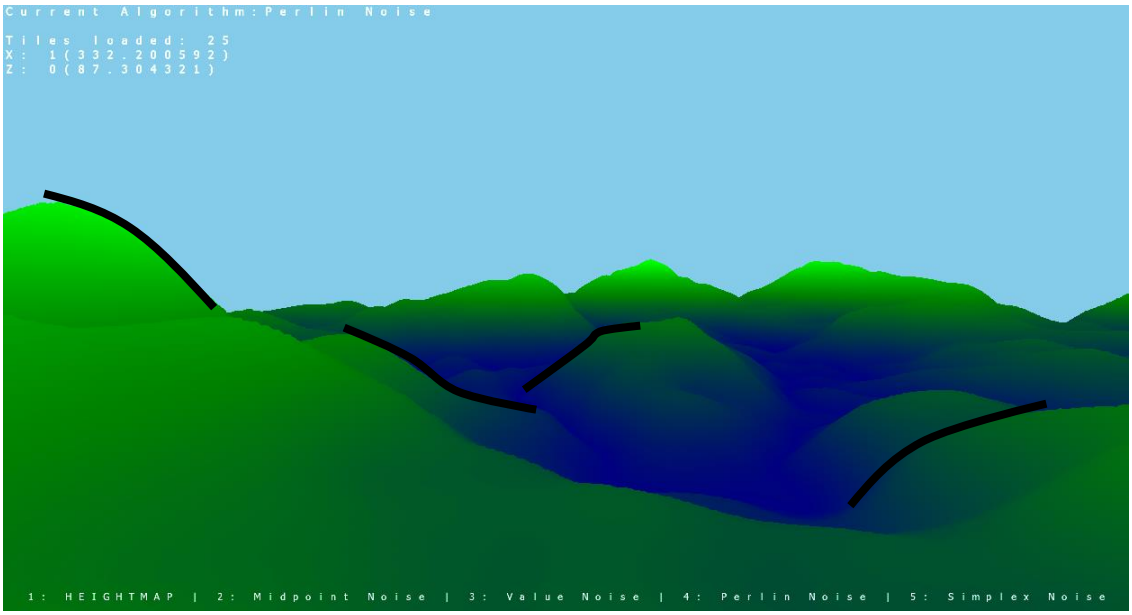


Figure 44: Terrain generated from using Perlin Noise.

4.3.4.5 Simplex Noise

As expected Simplex Noise is very close to Perlin Noise, it is after all built upon it.

Just like Perlin, it does not feature the smooth transitions that affect the Value Noise technique, therefore the steep slopes are present. The large variation between the minimum and maximum values is here, as shown by both green and blue terrain being drawn.

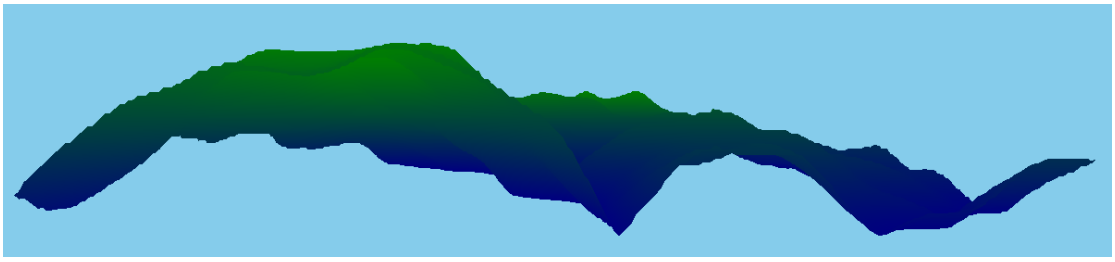


Figure 45: Orthographic view of Simplex Noise.

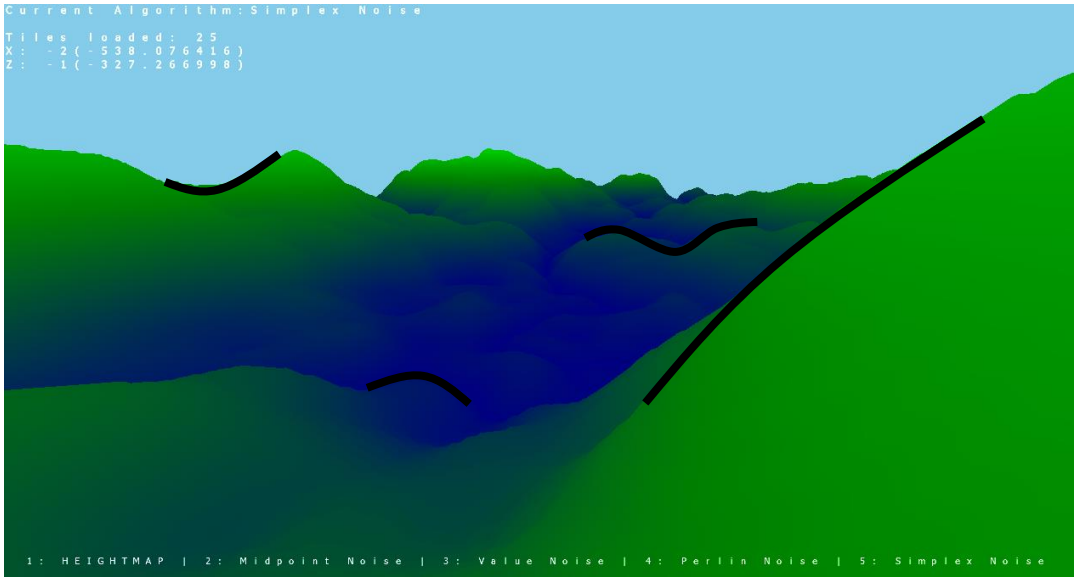


Figure 46: Terrain generated from using Simplex Noise.

4.4 Summary

Throughout this section, we've covered the performance of each technique from the FPS to RAM usage of each, finally looking at the actual terrain that is generated as a result. It was important to take this into account as even though this dissertation is looking at the performance of different technique, if there is only a couple or maybe one technique that outputs the desired terrain then performance goes out the window.

It has been show that whilst using a heightmap does result in very high and stable FPS the terrain itself is nothing to shout home about, the same can be said about the diamond-square technique the terrain it generates whilst it looks good in a standalone situation, it does not influence the surrounding tiles which results in an unnatural looking environment.

Value can be in a group on its own, whilst it does just like Perlin and Simplex generate natural and deterministic terrain, it introduces its own issues, such as the output range being low. Perlin and Simplex are then competing against one another.

4.5 Further testing

So that we can further back up the results and what has been said in the above summary an additional test was ran. This test is much like the one ran over all the techniques but with a more complex script and only run on Simplex and Perlin implementations.

The AutoHotKey script used is visible in Appendix A just as before. Results collection have been done in the same way as mentioned at the start of this chapter.

The idea behind this test is to change what is being rendered on screen more than just what we are looking at and the users position. This fits into the previous section in which the use of indices is addressed [3.2 Planning - 3.5.2 Indexing/Indices] and the tile system is discussed [3.2 Planning - 3.5.3 Tile system].

So far, the tile size has been fixed at 256 * 256 as previously mentioned that's 65,536 vertices and 393216 indices to render. This test is focusing on changing this value, both increasing it and decreasing the value, and evaluating the FPS that is achieved.

4.5.1 Change in tiles size being generated.

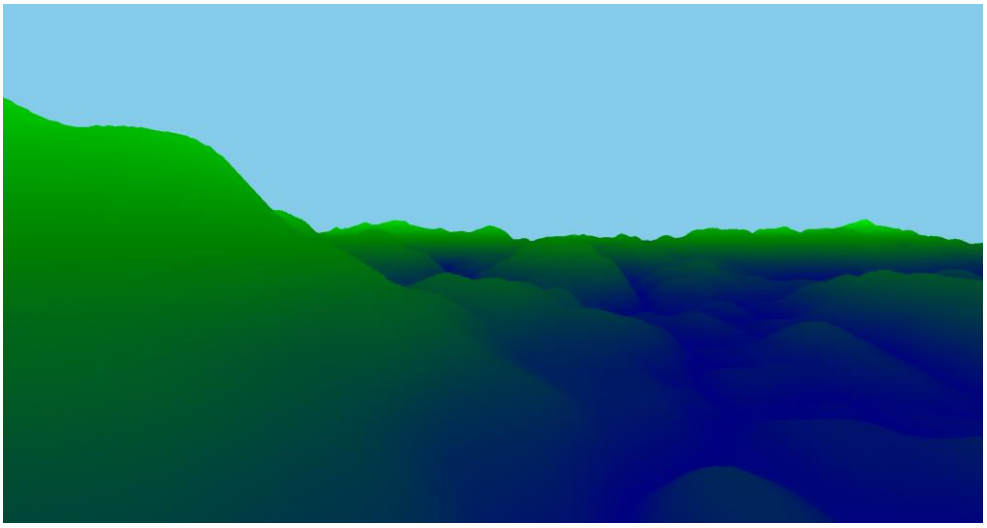
The first test ran is going to double the size of each tile. That will see the underlying system go up from 257 * 257 to 513 * 513, that's 263,169 vertices and a staggering 1,572,864 indices.

As has already been shown the terrain manager developed keeps the RAM usage constant meaning that the only impact this change will have is that the RAM usage will be increased but importantly at a constant level throughout the testing, hence why the focus is on the FPS.

```
25 in memory
```

Figure 47: Console output showing that the number of tiles hasn't increased.

Addressed in the summary section above the best techniques has shown to be Simplex Noise & Perlin Noise. These two have shown to generate the more natural looking terrain of all the techniques, whilst also achieving a high average FPS over the test script.



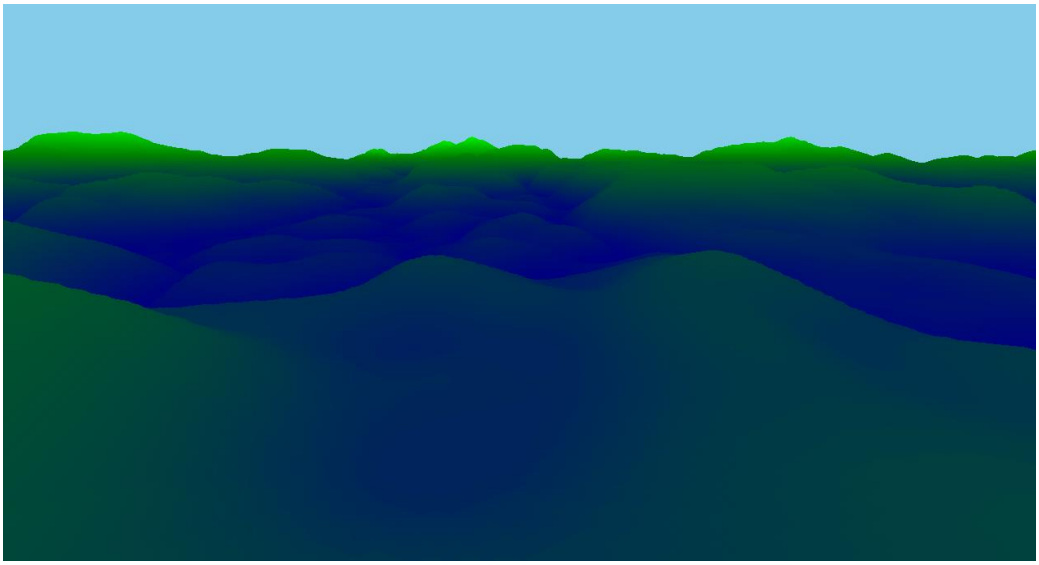


Figure 48: Simplex & Perlin terrain is shown to be the same as larger tile size.

As expected it is important to note that the terrain itself hasn't changed due to the change in size of the tiles underneath, further validating that the tool has been developed in such a way to provide a consistent testing environment. The same is said about smaller tiles.

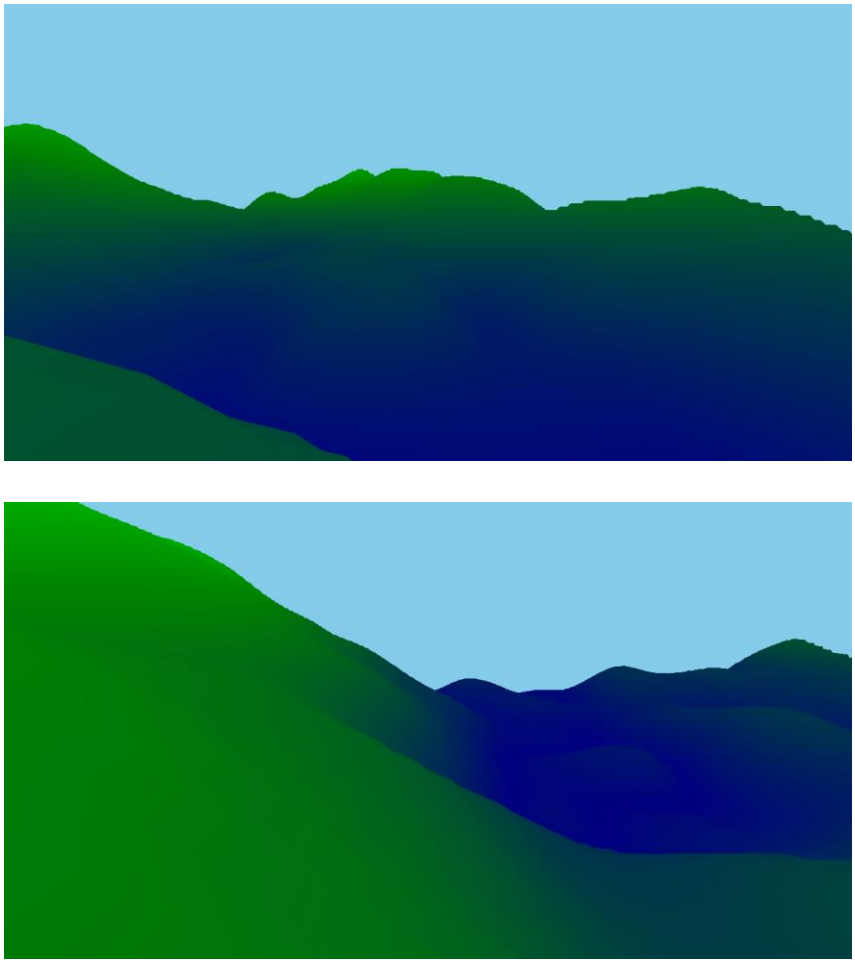


Figure 49: Simplex & Perlin smaller tiles.

The figures below show the FPS that the tool achieved whilst running the tests. Just like the previous test the two are very close to each other, especially in the case of the larger tiled system. The average FPS achieved is lower just as expected due the nature of more indices to process, and when the tile size was decreased a higher FPS was achieved, but once again in very close proximity to each other, with Perlin Noise pulling just 6 frames a second on average.

	FPS		
Technique	**Min**	**Max**	**AVG**
Simplex	**0**	**60**	**48**
Perlin	**0**	**59**	**48**

Table 2: FPS data from FRAPS when running a increased tile size.

	FPS		
Technique	**Min**	**Max**	**AVG**
Simplex	**410**	**717**	**640**
Perlin	**473**	**707**	**646**

Table 3: FPS data from FRAPS when running a decreased tile size.

The test has failed to separate out the two techniques. So that this can be addressed a timing system was implemented into the tool. With this done the time it took to generate each tile on a millisecond scale could be calculated. The full results of which are visible in Appendix D.

The timing system used the built in GameTimer class that's implemented into nclgl, this allowed access to a function that returned the amount of time since it was called, to get the time it took to generate a tile the tool makes two calls to this new function, one before our tile call and one after. Resulting in the after value being the time it took to generate a single tile. This test ran on the larger tile size of 513 * 513.

	Perlin Timings		Simplex Timings
Total Time	3.295532		3.1468725
Average Time	0.13182128		0.1258749

Table 4: Perlin v Simplex timing results.

The timings show that Simplex Noise in terms of the average time per tile generation and total time for all tiles on load is quicker than Perlin Noise. Proving the authors hypothesis correct.

Chapter 5 – Conclusion

This section presents a conclusion based on the aims and objectives that were outlined in chapter 1 [1.3 Project Aims and Objectives]. The conclusion will be based on whether the objectives have been met and as a result the overall aim.

An overview of the original objective will be given and an evaluation of that objective undertaken and possible further work that could be carried out in this research area is briefly introduced and touched upon.

5.1 Conclusion

5.1.1 Satisfaction of the Aims and Objectives

The overall aim of this dissertation was to carry out performance analysis of different terrain generation techniques, this was then broken down into objectives which were to be met so that the aim could be met. The completeness of these objectives will form the base of the conclusion.

1: 'To identify 5 generation techniques used by the industry today.

The developed tool does indeed incorporate techniques which are used with the industry to some extent the first set below have met this target.

- Heightmaps are very present within the video games industry. The latest Unreal Engine, Unreal 4 states in the documentation that,

  > *"Sometimes your Landscape will require that you use external programs to create both the height map and layers that are needed. Unreal Engine 4 (UE4) does accommodate this style of workflow"* [39]

- Diamond Square is a technique that has been referred to countless times as a method of generating heightmaps.

  > *"Heightmaps are a representation of a three-dimensional world as could be generated by the diamond-square algorithm."* [40]

  It has already been established that Heightmaps are used so this is the perfect partner to fully explore the Heightmap technique.

- Perlin Noise is used in the largest open world sandbox game today. Minecraft. Back when Markus Persson owned Mojang the company behind Minecraft, he wrote a post describing the terrain generation.

  > *"I used a 2D Perlin noise heightmap to set the shape of the world"*
  > *"system based off 3D Perlin noise"* [41]

However, Value Noise and Simplex noise are not as commonly used as the others, but these have been included as they relate to Perlin Noise and in the case of Simplex is an advancement on Perlin Noise and for Value it's because it's a non-gradient noise function. Resulting in a different output.

2: 'To develop a tool, in which the 5 techniques can be ran and compared.'

The tool has been developed in such a way that when switching between techniques the only aspect that changes is the technique / algorithm that is being used to generate the terrain values. Everything from the structure of the terrain data and the infinite world system to the way the data is rendered is identical from technique to technique.

3: 'Implement the different techniques into the tool.'

The has fully been met. The developed tool is capable of switching between the techniques outlined as part of objective 1 on the fly, repeatable without hassle. The terrain gets regenerated around the player with the chosen technique.

4: 'Evaluate the techniques.'

A performance analysis was undertaken on all of the mentioned techniques, this included looking at the FPS achieved during a pre-defined testing solution, the RAM usage of each technique, GPU usage and the overall terrain that is produced from each of the techniques. Therefore, is was met and subsequently the overall aim of the project has been met.

> This dissertation aims to evaluate different techniques used to generate terrain procedurally in real-time simulations.

5.1.2 What Has Been Established and What could be done further.

The hypothesis of this project as written in the abstract states that the best technique to use for natural looking real-time terrain generation is to use Simplex Noise and as stated in chapter 4s' summary the tool developed has shown this to be correct.

Due to the following:

- Simplex does not suffer from the smooth changes in output values as Value does.
- Simplex conforms to the pseudo-randomness, unlike the Diamond Square technique.
- Simplex uses the full output spectrum.
- Simplex does not repeat, such as a Heightmap would.
- Simplex has faster generation over Perlin Noise.

5.1.3 What went well

Overall the author believes that the project went well. The research into different techniques, the developed system and sub-systems within it such as the tiling system to allow for the infinite world was best designed for the situation at hand. The use of an underlying framework to ensure that the testing was fair and comparable, along with the testing methodology was key to the success of this project.

5.1.4 What could have been done better

The outcome of the project and the validation of the hypothesis is solely based upon the authors point of view, because of this the accuracy of the results could be questioned. In order to further back-up the results, external user feedback could have been gathered, this would have been a good resource to have as for example how natural some terrain looks is very user preference heavy, that is that each user would have a different opinion.

5.1.5 What could be done in the future

So that the author can further back up the conclusion of this dissertation additional steps could be undertaken. As has been mentioned throughout this dissertation Simplex Noise was originally created to address issues in the original Perlin Noise algorithm whilst also providing speed improvement in the higher dimensions. This is an area which could be investigated further. The comparison between generation times of a 4D or 5D Perlin Noise and a 4/5D Simplex Noise implementation.

References

[1] D. Takahashi, "PwC: Game industry to grow nearly 5% annually through 2020," VentrueBeat, 2017. [Online]. Available: htps://venturebeat.com/2016/06/08.the-u-s-and-global-game-industries-will-grow-a-healthy-amount-by-2020-pwc0-forecasts/. [Accessed 27 03 2017].

[2] A. Stewart, "Skyrim is Coming to the Nintendo Switch," 2017. [Online]. Available: http://realgamemedia.com/skyrim-coming-nintendo-switch/. [Accessed 27 03 2017].

[3] N. Llopis, "Procedural Content Creation," in *GameDeveloper The leading game industry magazine*, Game Developer, 2009, pp. 41-43.

[4] Brightman, "Games must achieve photorealism in order to open up new genres says 2K," GamesIndustry.biz, 2012. [Online]. Available: : http://www.gamesindustry.biz/articles/2012-08-01-games-must-achieve-photorealism-in-order-to-open-up-new-genres-says-2k . [Accessed 22 02 2017].

[5] B. J and V. P. P. J-N, "The What, Where and Why of Real-TIme Simulation," Opal-RT Technologies, Unknown.

[6] E. S, "Nothing random about it," Quel Solaar, 01 04 2010. [Online]. Available: http://news.quelsolaar.com/#post70. [Accessed 29 03 2017].

[7] J. Lee, "How Procedural Generation Took Over The Gaming Industry," 26 11 2014. [Online]. Available: http://www.makeuseof.com/tag/procedural-generation-took-gaming-industry/. [Accessed 04 05 2017].

[8] "Procedural Content Generation Wiki," 22 08 2010. [Online]. Available: http://pcg.wikidot.com/pcg-games:borderlands. [Accessed 04 05 2017].

[9] "Procedural Content Generation Wiki," 18 09 2011. [Online]. Available: http://pcg.wikidot.com/pcg-games:spore. [Accessed 04 05 2017].

[10] Greg, "No Man's Sky - Procedural Content," 3dgamedevblog, 27 10 2016. [Online]. Available: http://3dgamedevblog.com/wordpress/?p=836. [Accessed 29 03 2017].

[11] Unknwon, "The Overworld," gamepedia.com, 10 03 2017. [Online]. Available: http://minecraft.gamepedia.com/The_Overworld. [Accessed 28 03 2017].

[12] M. G, "The Definition and Rendering of Terrain Maps," Cambridge University Engineering Department, Cambrdige, 1986.

[13] http://www.redblobgames.com/articles/noise/introduction.html, "Noise Functions and Map Generation," Red Blob Games, 31 08 2013. [Online]. Available: http://www.redblobgames.com/articles/noise/introduction.html. [Accessed 25 03 2017].

[14] Unknown, "eskipaper.com," eskipaper, [Online]. Available: http://eskipaper.com/snow-mountain-8.html#gal_post_68875_snow-mountain-8.jpg. [Accessed 29 03 2017].

[15] Scratchapixel, "Value Noise and Procedural Patterns: Part 1," 15 05 2016. [Online]. Available: https://www.scratchapixel.com/lessons/procedural-generation-virtual-worlds/procedural-patterns-noise-part-1/creating-simple-1D-noise. [Accessed 09 11 2016].

[16] K. P, "SIGGRAPH 2002 Course 36 Notes: Chapter 2 - Noise Hardware," SIGGRAPH, 2002.

[17] A. M, "Optimizations in perlin noise-generated procedural terrain," Department of Computer Science, Babes¸-Bolyai University, Kogalniceanu, 2012.

[18] leonbloy, "math.stackexchange," 19 08 2012. [Online]. Available: https://math.stackexchange.com/questions/184121/why-is-gradient-noise-better-quality-than-value-noise. [Accessed 30 0 2017].

[19] R. L. Cook and T. DeRose, "Wavelet Noise," Pixar Animation Studios, Unknown.

[20] A. F, "My Global IT," ?? 01 2013. [Online]. Available: http://www.myglobalit.com/blog/waterfall-vs-agile-models-software-development. [Accessed 30 03 2017].

[21] "Aglie vs Waerfall," Agile In A Nutshell, Unknown. [Online]. Available: http://www.agilenutshell.com/agile_vs_waterfall. [Accessed 30 03 2017].

[22] "gamedev.stackexchange," stackexchange, 20 09 2012. [Online]. Available: https://gamedev.stackexchange.com/questions/37389/diamond-square-terrain-generation-problem. [Accessed 03 31 2017].

[23] knight666, "gamedev.stackexchange," stackexchange, 05 07 2012. [Online]. Available: https://gamedev.stackexchange.com/questions/31668/why-is-c-c-preferred-for-game-developers. [Accessed 28 03 2017].

[24] Unknown, "OpenGL Overview," OpenGL, Unknown. [Online]. Available: https://www.opengl.org/about/. [Accessed 31 03 2017].

[25] N. G. E. Lecturers, "Introduction to the Graphics Module Framework," 2015. [Online]. Available: https://research.ncl.ac.uk/game/mastersdegree/graphicsforgames/introductiontonclgl/Introduction%20To%20Framework.pdf. [Accessed 31 03 2017].

[26] J. Norris, "Fraps Review," Pc Advisor, 19 01 2012. [Online]. Available: http://www.pcadvisor.co.uk/review/disk-tools-software/fraps-review-3331211/. [Accessed 31 03 201].

[27] Unknown, "Afterburner," MSI, 2017. [Online]. Available: https://www.msi.com/page/afterburner. [Accessed 30 03 2017].

[28] Unknown, "Tutorial 9: VBO Indexing," opengl-tutorial, Unknown. [Online]. Available: http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-9-vbo-indexing/. [Accessed 03 04 2017].

[29] N. G. E. Lecturers, "Tutorial 8: Index Buffers & Face Culling," Unknown. [Online]. Available: http://research.ncl.ac.uk/game/mastersdegree/graphicsforgames/indexbuffers/Tutorial%208%20-%20Index%20Buffers.pdf. [Accessed 03 04 2017].

[30] Jimmy, "Diamond-square terrain generation problem, [2]," 20 09 2012. [Online]. Available: https://gamedev.stackexchange.com/questions/37389/diamond-square-terrain-generation-problem/37421. [Accessed 08 12 2016].

[31] C. Randall, "Code: The Diamond Square Algorithm," BLuh.org, 31 01 2012. [Online]. Available: https://www.bluh.org/code-the-diamond-square-algorithm/. [Accessed 18 11 2016].

[32] L. Benstead, "A Public Domain C++11 1D/2D/3D Perlin Noise Generator," 26 05 2014. [Online]. Available: http://blog.kazade.co.uk/2014/05/a-public-domain-c11-1d2d3d-perlin-noise.html. [Accessed 29 01 2017].

[33] S. Gustavson, "Simplex noise demystified," 22 03 2005. [Online]. Available: http://webstaff.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf. [Accessed 24 10 2016].

[34] Eevee, fuzzy notepad - Perlin noise, 29 05 2016. [Online]. Available: https://eev.ee/blog/2016/05/29/perlin-noise/. [Accessed 29 03 2017].

[35] K. Perlin, "Standard for perlin noise". Patent US6867776 B2, 08 01 2002.

[36] Flafla2, "Understanding Perlin Noise," 09 08 2014. [Online]. Available: https://flafla2.github.io/2014/08/09/perlinnoise.html. [Accessed 04 04 2017].

[37] Informatix, "Simplex Noise," b4x, 03 01 2013. [Online]. Available: https://www.b4x.com/android/forum/threads/simplex-noise.36315/. [Accessed 05 04 2017].

[38] Unknown, "AutoHotKey," Unknown. [Online]. Available: https://autohotkey.com/. [Accessed 05 04 2014].

[39] Unknown, "Creating and Using Custom Heightmaps and Layers," EPIC GAMES, Unknown. [Online]. Available: https://docs.unrealengine.com/latest/INT/Engine/Landscape/Custom/. [Accessed 11 04 2017].

[40] U. o. Twente, "Challenges in Procedural Terrain," University of Twente, Enschede, 2010.

[41] M. Persson, "Terrain generation, Part 1," 9 03 2011. [Online]. Available: https://notch.tumblr.com/post/3746989361/terrain-generation-part-1. [Accessed 11 04 2017].

Appendix A

AutoHotKey Script used for initial testing.

```
$p::
	Send {w down}
	sleep 300000 		; +5 mins

	Send {w up}

	Send {w down}
	Send {a down}
	sleep 300000 		; +5 (10 mins)


	Send {w up}
	Send {a up}

return
```

AutoHotKey Script for the second testing with larger/smaller tiles.

```
$p::
	Loop, 10 {

		Send {w down}
		sleep 3000 		; +3 seconds

		Send {a down}
		sleep 1000 		; +4
		Send {a up}

		Send {k down}
		sleep 250 		; +4.25
		Send {k up}

		Send {a down}
		sleep 250 		; +4.5
		Send {a up}

		Send {s down}
		sleep 250 		; +4.75
		Send {s up}

		Send {d down}
		sleep 250 		; +5
		Send {d up}

		Sleep 10000 		; +15 seconds
		Send {w up}

		Send {l down}
		sleep 2500 		; +17.5
		Send {l up}

		Send {w down}
		sleep 12500 		; +30 seconds
		Send {w up}
	}

return
```

In both situations, when the script is active, when the user presses the P button the script begins.

Send{ ? down } –

This sends a button-down command to the operating system; it acts as if the button is being held down.

Send{ ? up } –

This send a button-up command to the operating system, acts as if the button has been released.

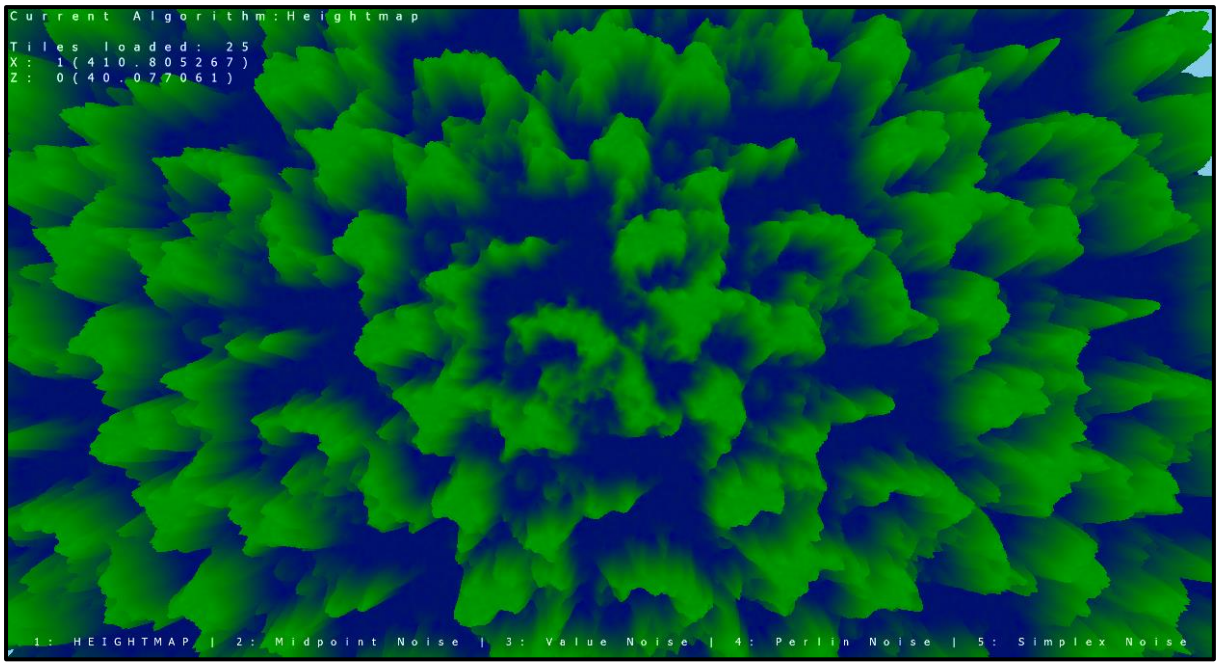
Sleep ????? –

Puts the script to sleep at the current location of execution.

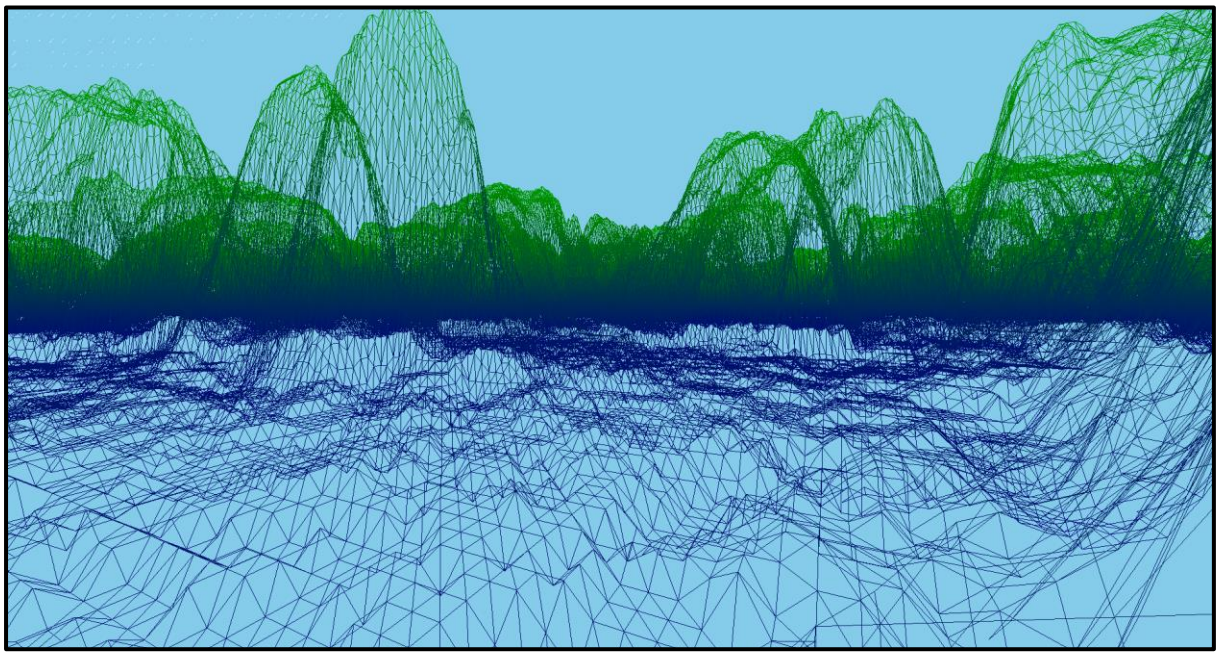
Appendix B

Terrain screenshots

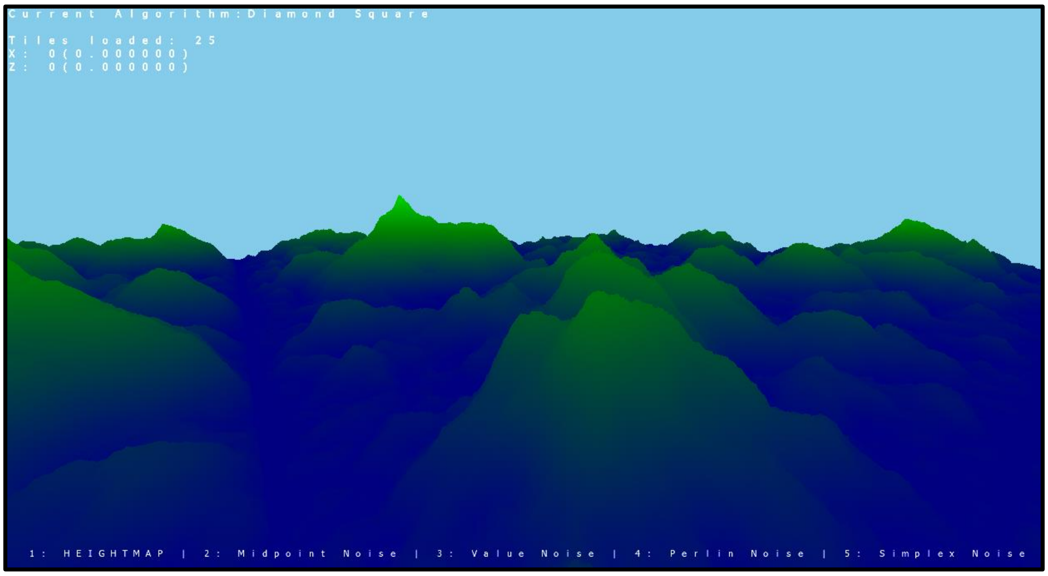
Heightmap

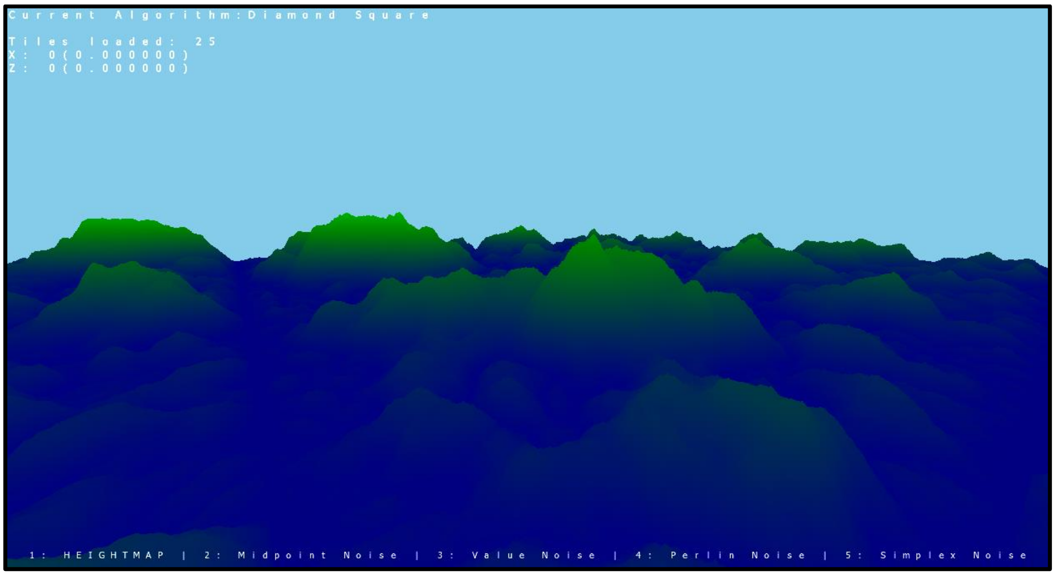


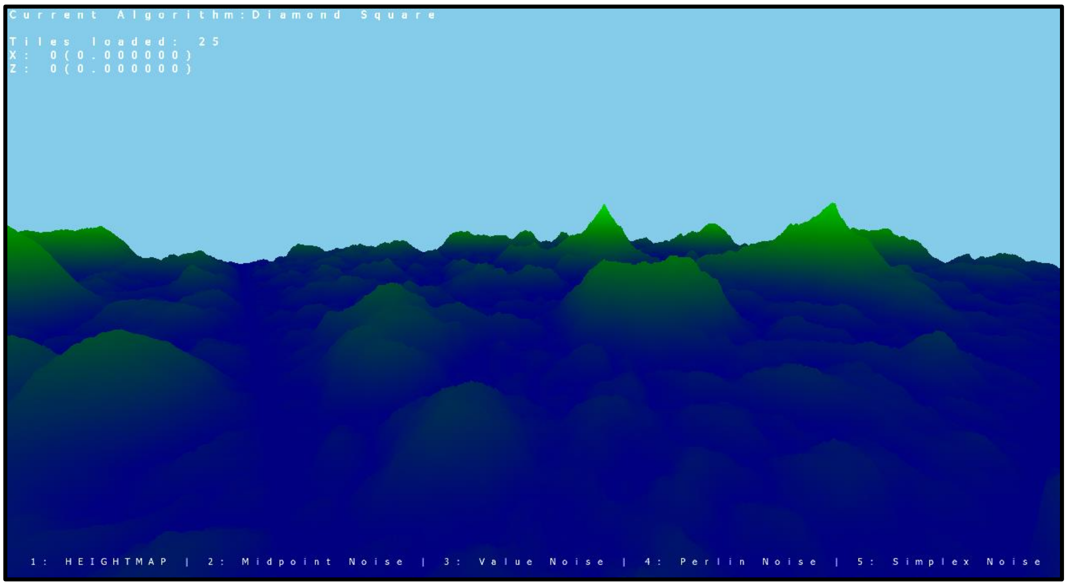
Above is a prime example of the terrain from the Heightmap technique repeating over every tile.

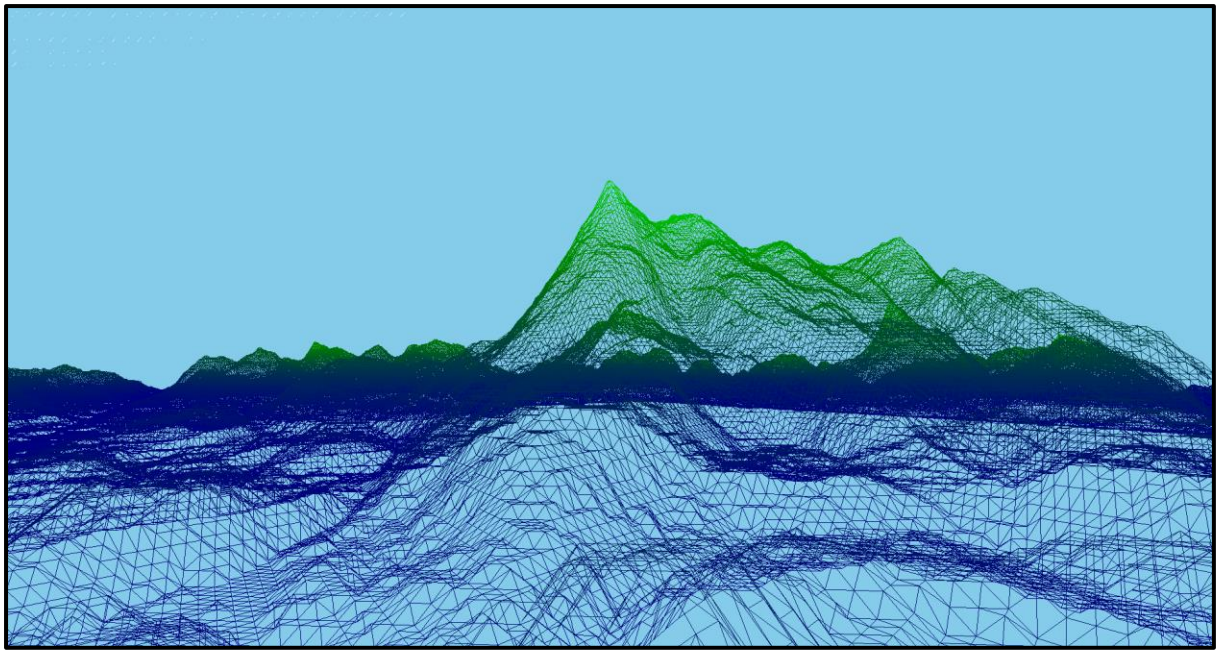


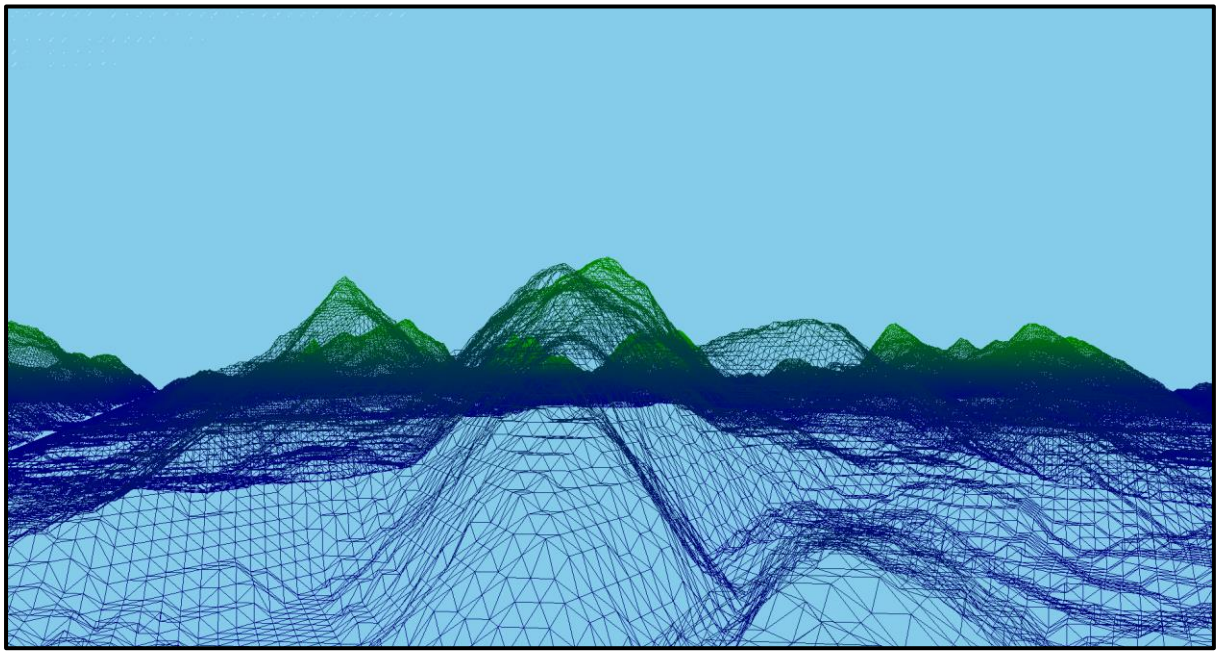
Diamond-Square





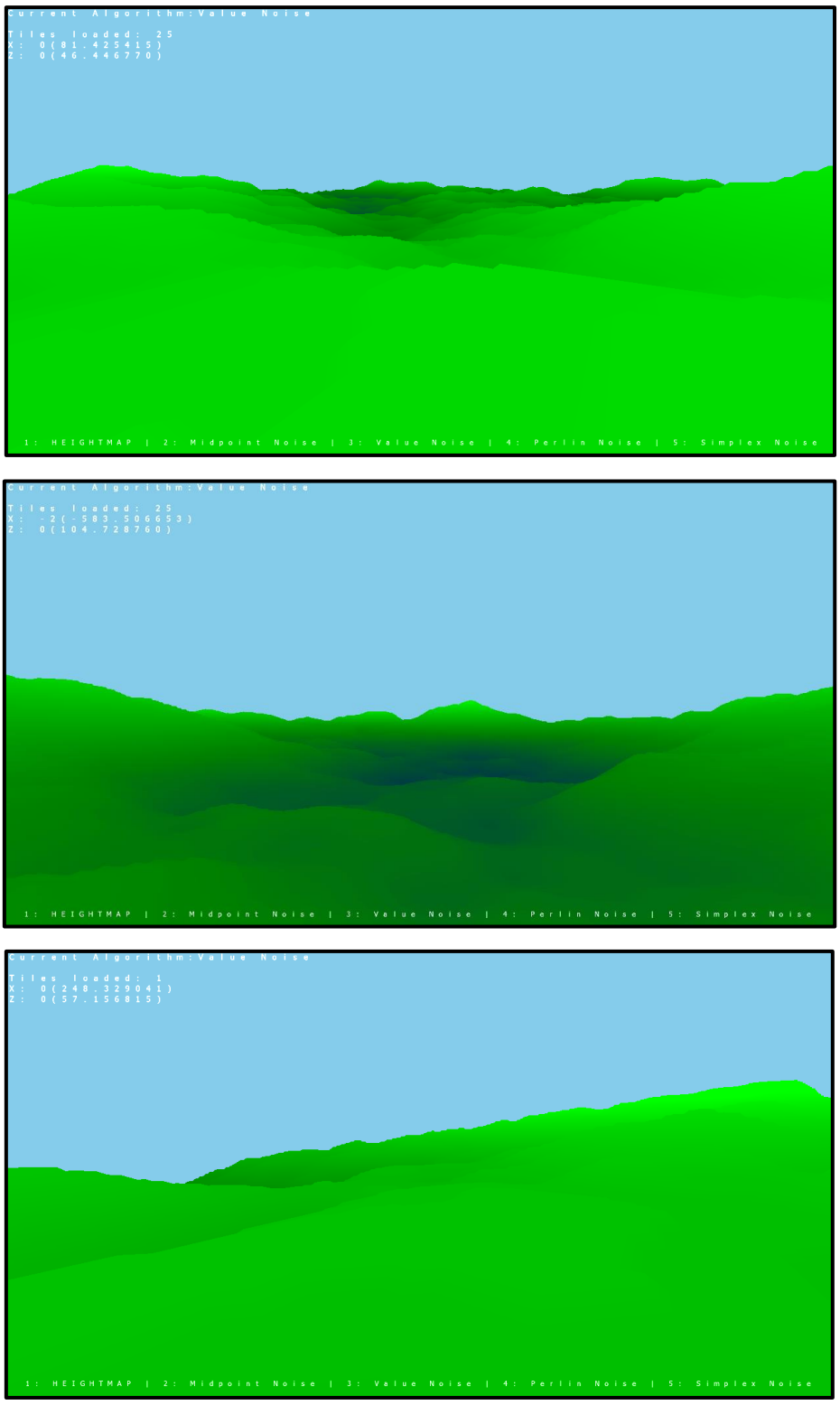




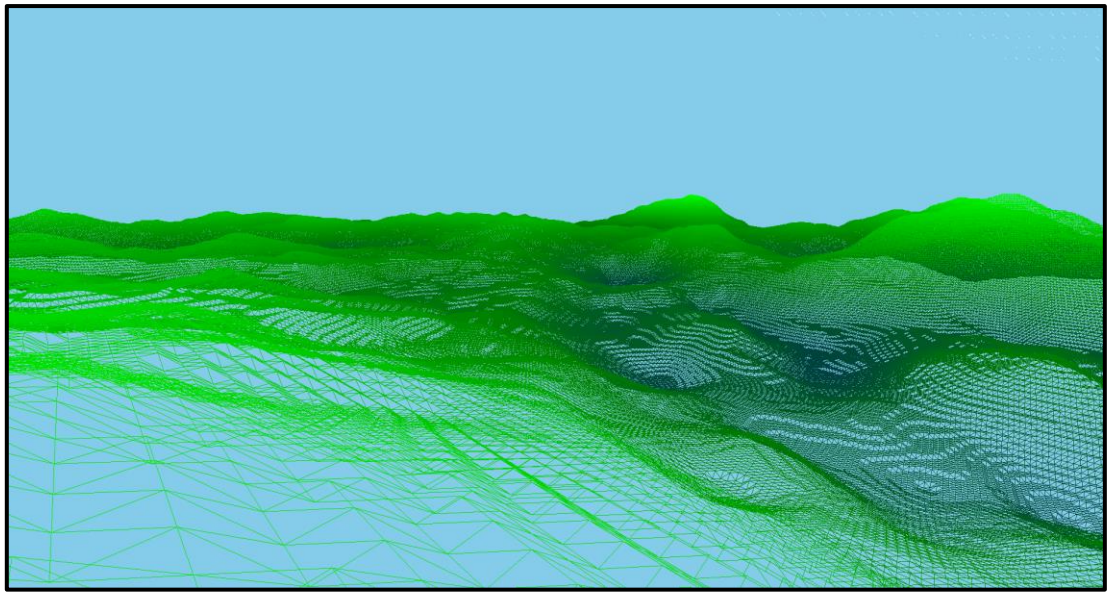


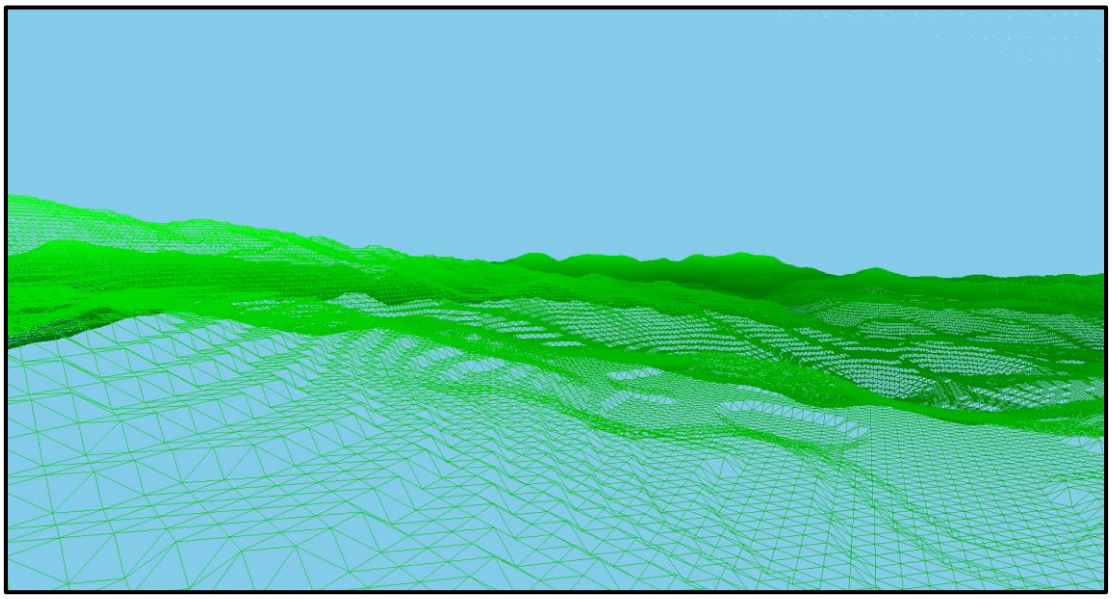
In the figures above the terrain generated is all in the same location, and yet the terrain itself is different, this is a prime example of how the diamond-square technique is not pseudo-random.

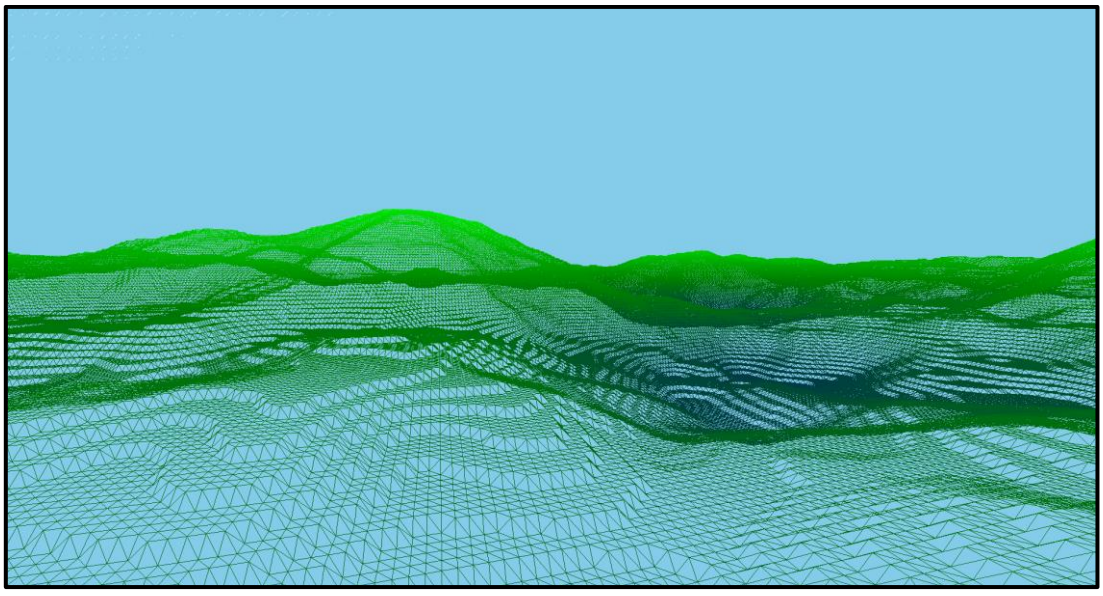
Value Noise



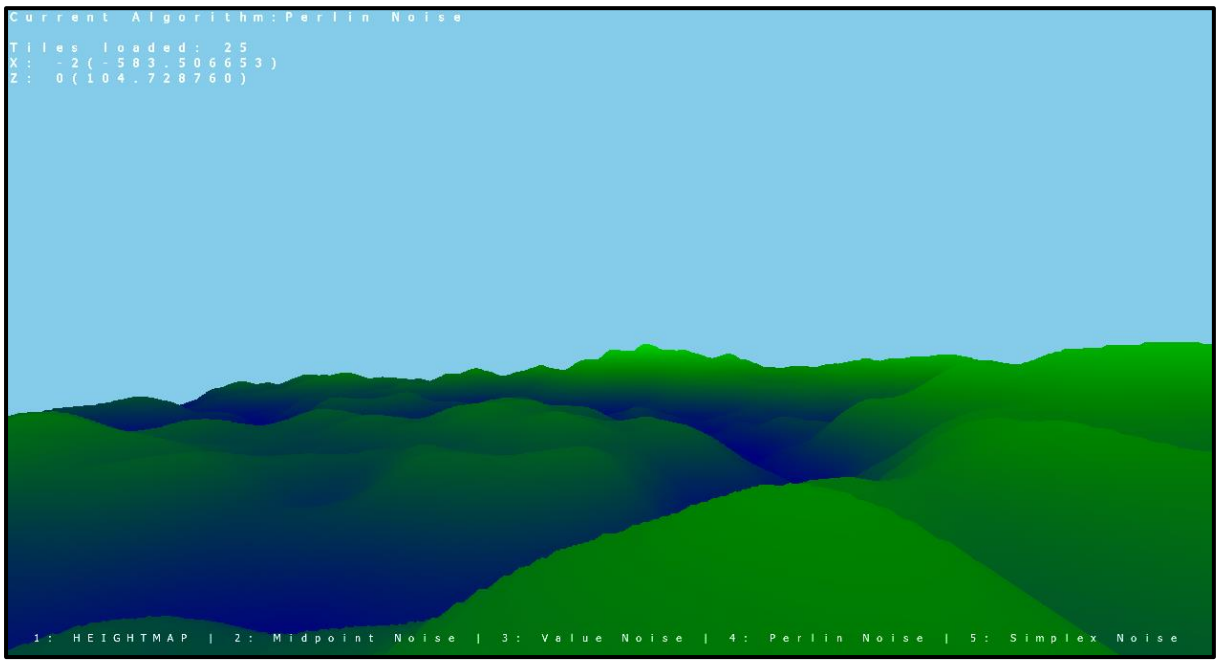
As can be seen above the smooth slopes generated from Value Noise stretch for the entire rendered landscape.

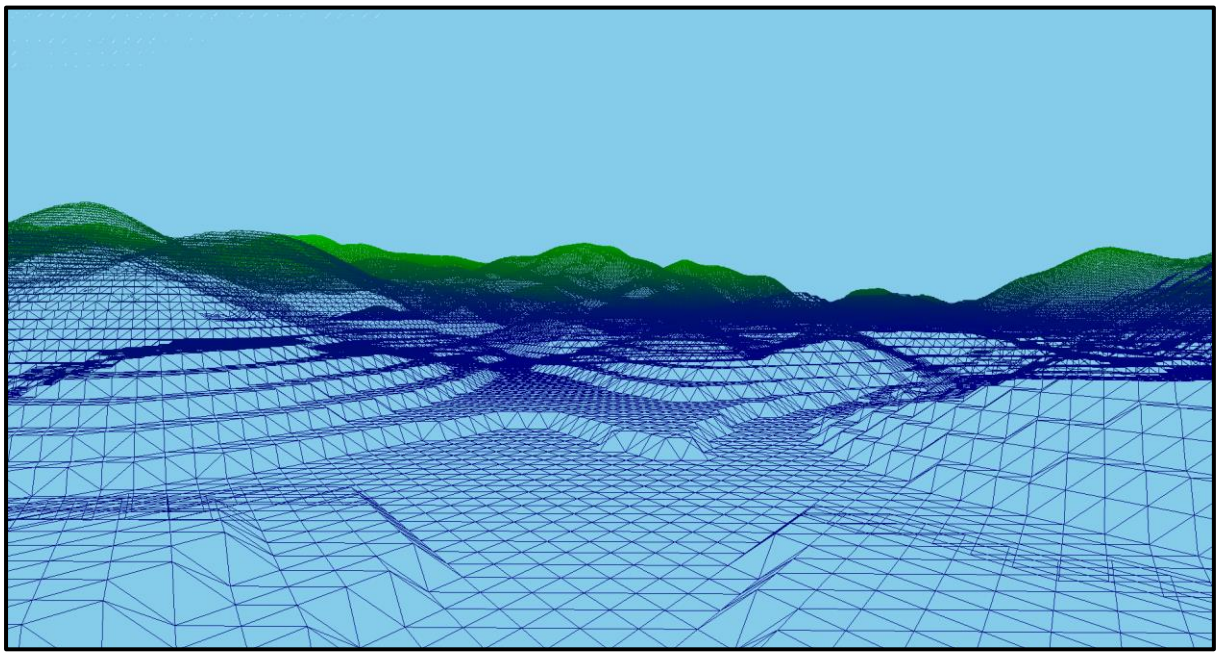






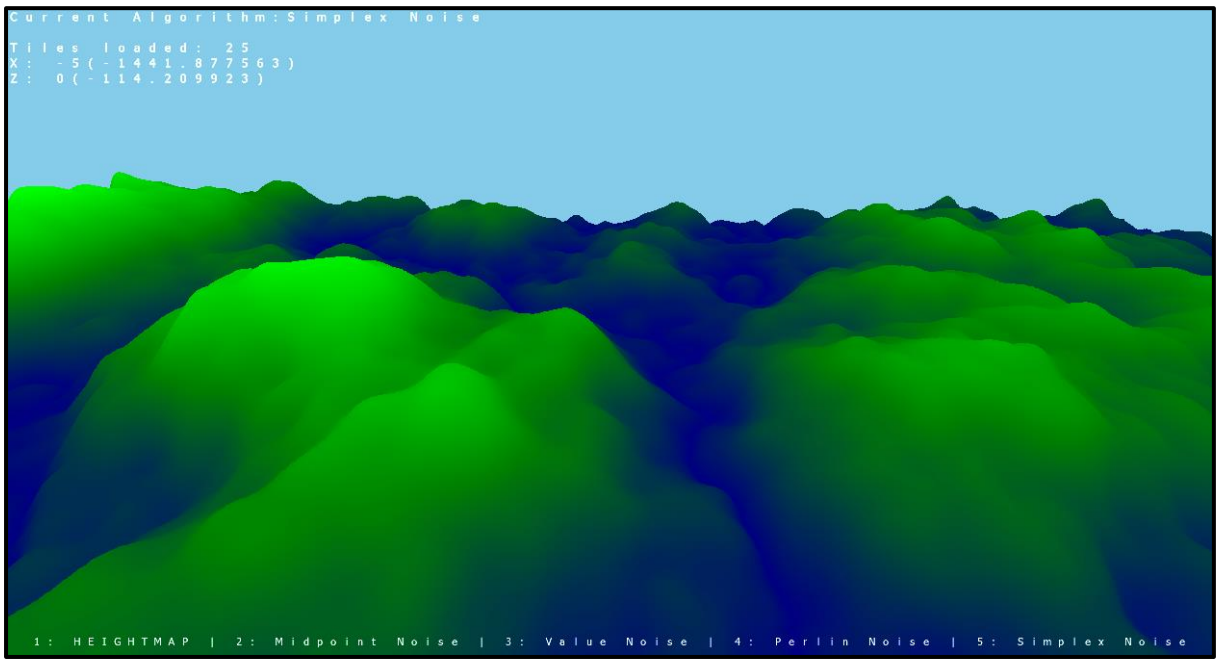
Perlin Noise

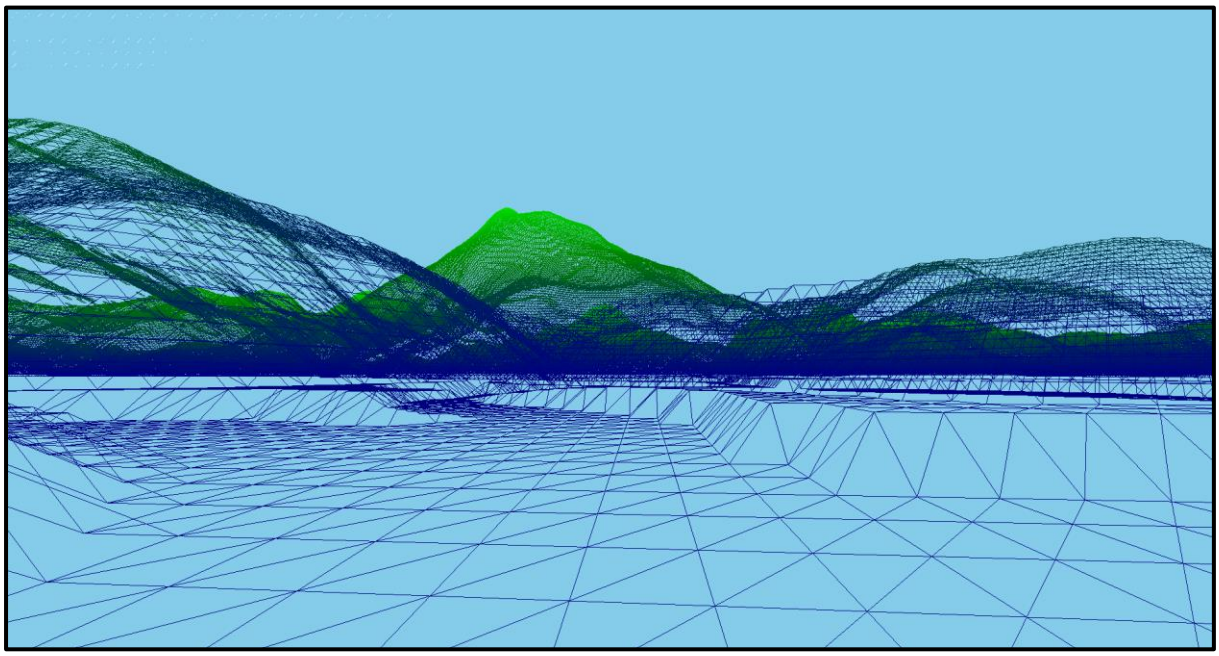




However, in the figures above and below its shown that both Perlin and Simplex Noise techniques do not suffer from this effect, resulting in the terrain generated is far more natural looking.

Simplex Noise





Appendix C

Fragment shader code used for terrain colouring.

```
void main(void) {

	float heightvalue = (IN.pos.y / 100.0f);

	float green = heightvalue;
	float blue = 0.5 - green;

	FragColour = vec4(0.0f, green, blue, 1.0f);
}
```

Appendix D

Full results from timing testing of Perlin and Simplex Noise

	Perlin Timings		Simplex Timings
	165.587		121.484
	149.486		107.953
	134.736		99.5
	120.019		104.25
	128.802		103.594
	150.361		113.891
	149.32		122.078
	124.528		103.703
	137.462		110.453
	134.509		122.594
	124.292		102.297
	120.78		99.6719
	141.989		106.75
	146.727		152.266
	108.382		181.125
	106.753		147.656
	102.383		99.0156
	132.855		141.188
	119.231		141.153
	135.451		146.797
	125.848		146.797
	115.246		153.047
	146.317		156.734
	139.519		127.984
	134.949		134.891
Total Time	3.295532		3.1468725
Average Time	0.13182128		0.1258749

Glossary

Acronym	*Expansion*	*Explanation*
.push_back		Function in C++ to add an object to the end of a vector.
AHK	AutoHotKey	Windows application that allows users to automate repetitive task in windows.
API	Application Programming Interface	Set of functions and procedures that allow the creation of applications from the features from other applications.
big-O		Theoretical measurement of an algorithm's execution time.
CPU	Central Processing Unit	The “brain” of any computer. Sends signals to control other parts of the computer.
DirectX	Microsoft DirectX	Collection of APIs for handling multimedia tasks
FOV	Field of View	Extent of the observable world that is seen at any given time.
FPS	Frames Per Second	Measurement of how many times the screen is refreshed/redrawn each second.
GL_TRIANGLES	Primitive type in OpenGL	Tells OpenGL that vertex 0,1,2 form a triangle, vertex 3,4,5 another and so on.
GPU	Graphics Processing Unit	Processor whose task is mainly calculating the graphical output for monitors.
GUI	Graphical User Interface	Set of graphics such as buttons to allow a user input.
IDE	Integrated development environment	Software application used to develop software applications.
Lattice	Lattice (group)	A repeating arrangement of points.
NCLGL	Newcastle Game Library	Newcastle University game framework.
OOP	Object Orientated Programming	A model for writing computer programs which uses objects that interact with each other instead of a list of actions to be carried out one after another.
OpenGL	Open Graphics Library	Cross-languages, cross-platform API for rendering graphics.
OS	Operating System	System software that manages computer hardware and software resources.
PRN	Pseudorandom number generator	A number generator, where the outcome will be the same for the same initialise number.
RAM	Random Access Memory	Part of the storage solutions in a computer. Information is accessible quickly in any order.
RAW	File type	A file type where the data is unprocessed. Often used by software developers.