Analysis of the Performance and Effectiveness of Techniques for Real-Time Terrain Generation

Bailey Gardner (160337608)

BSc Computing Science

Project Supervisor: Dr Graham Morgan

Word Count: 14752

May 2019

Abstract

This dissertation aims to compare a variety of different procedural terrain generation techniques to conclude which is the most suitable for use, depending on user requirements.

Research carried out into each technique is described followed by the steps taken to implement and ultimately test the techniques by comparing and evaluating their performance, system resource requirements and visual output. A final argument is then presented detailing which techniques are most suitable based on the usage situation.

Declaration

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

Acknowledgements

I would like to thank my supervisor Dr Graham Morgan as well as Dr Gary Ushaw, Dr Rich Davison and Dr Neil Speirs for their continued support by providing invaluable guidance and feedback throughout this project.

Table of Contents

Table of Figures

Chapter 1. Introduction

1.1 Purpose

The video game industry is bigger than ever before and is seemingly going to continue to grow at a rapid rate for the foreseeable future. This places a huge responsibility on video game developers to create bigger and better games with each successive release.

The problem with this is while expectations for the size and scale of video games are constantly increasing, development time usually is not. As a result, developers may need to rely on clever techniques which help automate, to a high standard, certain aspects of the development process. This is where terrain generation comes in.

Arguably one of the most time-consuming aspects of game design is creating a world in which the player can immerse themselves in. Frequently these worlds are created using some form of terrain. The problem with creating terrain manually is it can be very time-consuming however, it is also extremely difficult to manually create terrain in a way that it appears natural and, in some way, random. A good way to solve this problem is by generating terrain procedurally which simply means rather than manually creating each piece of terrain, algorithms can instead be used to quickly and effectively generate huge chunks of a game world.

Many algorithms exist that can be used for terrain generation which means it is important to understand the differences between them. This dissertation aims to highlight these differences by comparing some of the most commonly known algorithms in terms of their cost in performance but also in the effectiveness of the terrain that they can output.

1.2 Aim and Objectives

The aim of this dissertation project is:

> To compare and evaluate commonly used techniques for real-time terrain generation.

The objectives which aid in achieving this aim are:

- Investigate commonly used real-time terrain generation techniques.

This first objective will identify the techniques that will be present throughout this project.

- Implement an efficient method of creating meshes suitable for terrain generation.

This objective is very important as it entails the development of the base mesh that will be used to apply each terrain generation technique. The code for the mesh used for each technique will remain constant so that the performance and effectiveness of each algorithm can be measured as fairly as possible.

- Apply terrain generation techniques to a suitable mesh.

As already stated, only the algorithm used to generate heights for the mesh will change so that each algorithm can be compared as fairly as possible.

- Collect, compare and evaluate performance and effectiveness of the discovered techniques.

Suitable data will be collected to evaluate the performance of each algorithm for example, Frames Per Second (FPS). The effectiveness of each algorithm will be decided by a short survey designed to find which algorithm produces the most aesthetically pleasing terrain.

1.3 Dissertation Outline

Introduction

- Purpose
- Aims and Objectives
- Dissertation Outline

The introductory section sets the foundation of the dissertation by explaining the motivational reasoning behind the project as well as including the aim and description of objectives detailing how the project will be approached.

Background Material

- Procedural Content Generation
- Noise
- Map Generation

This section describes the research that went into procedural content generation as a whole before further describing the research acquired on every terrain generation technique used for this project.

Design and Implementation

- Software and Technologies
- Implementation

This section begins by describing the software and technologies used to create and test the code for this project. It then leads into the steps of how the project code was implemented.

Testing and Results Evaluation

- Testing Methodology
- Results
- Evaluating the Results

This section includes descriptions for the methods of how the terrain generation techniques were tested and what was tested for each. The results of these tests are then presented and

evaluated with the addition of a survey based on retrieving opinions on how the output of each technique compares.

Conclusion

- Project Aim
- What was Learnt
- Future Improvements

This section concludes the project by explaining if the project met the set objectives and in turn, the overall aim and if so, how this was achieved. This is followed by an explanation of what was learnt throughout the project and what could possibly be done in the future to improve it.

Chapter 2. Background Material

2.1 Procedural Content Generation

There are many techniques that allow for naturally shaped terrain to be generated within video games. Many of the commonly used techniques involve setting the height value of a vertex based on the position of their X and Z values however, many techniques achieve this in a variety of ways.

2.1.1 Online and Offline Generation

Online generation refers to content that is generated at runtime. This type of generation is very extensively used in games like Minecraft which have an almost infinite world. This is achieved by the world generating in real-time as the player walks around. As the player moves within range of the current world boundary a new procedurally generated chunk is created.

On the other hand, offline generation happens during the development of the game where a base model is generated by an algorithm which is then completed by artists and developers. "*An algorithm suggests interior layouts that are then edited and perfected by a human designer before the game is shipped*" [1]

This creates specific situations in which to use each type of technique. Online generation techniques require lightning fast execution as they are most commonly used for solutions to real-time problems where the application is usually updated every frame. For example, a game running at 60 FPS offers about 16 to 17 milliseconds(ms) between each frame. There are many calculations that must be carried out during this time, so it is vital that each calculation computes as quickly as possible. However, offline generation techniques have the privilege to take much longer which may allow them to provide much better results, specific to the needs of the developer.

Figure 1. Showcase of Online Terrain Generation in Minecraft [2]

2.2 Noise

In computer graphics noise is very commonly used to help create the appearance of randomness. When generating terrain in a video game it is important to ensure terrain appears as random as possible but also as natural as possible so that the player remains immersed within the world. To achieve this, functions named noise functions can be used.

2.2.1 Perlin Noise

Perlin noise is perhaps the most well-known noise function used for the generation of computer graphics. A simple Google search for video game terrain generation returns thousands of results explaining how Perlin noise can be used to generate pseudo-random, natural looking terrain. The problem, however, is Ken Perlin, the creator of Perlin noise, has iterated on his original creation to create more improved versions of the classic noise which make it difficult to find content based on classic Perlin noise since even the newer, improved versions also go by the name of Perlin noise.

"*Perlin Noise is a so-called gradient noise, which means that you set a pseudo-random gradient at regularly spaced points in space and interpolate a smooth function between those points.*" [3]

Created in 1983 by Ken Perlin, the classic Perlin noise technique was Ken Perlins' first development of a pseudo-random noise function [4]. This implementation is a gradient based noise function which can be used for much more than terrain generation as Perlin initially developed the technique for generating procedural textures.

Figure 3. Marble vase generated using classic Perlin Noise [6]

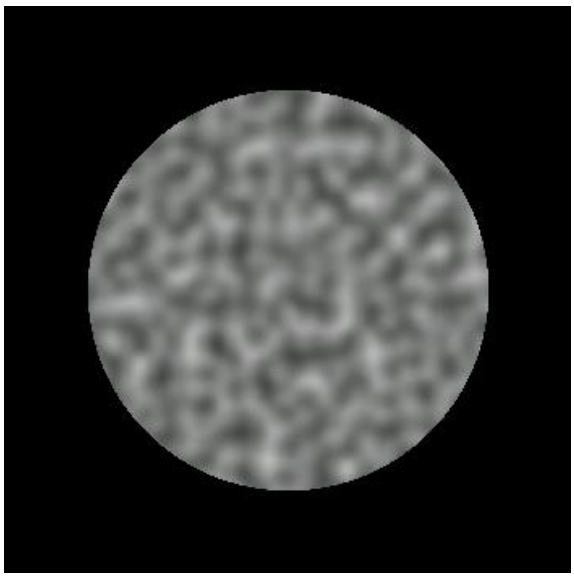


Figure 2. Classic Perlin Noise [5]

2.2.2 Simplex Noise

Simplex noise, created in 2001, is Ken Perlins' newer version of his classical Perlin noise. It was designed to produce fewer artefacts and to run more efficiently in terms of the scalability of the algorithm at higher dimensions.

"*A redesign of Perlin noise, also by Ken Perlin, simplex noise attempts to reduce the complexity of higher dimensional noise functions*" [7]

Like Perlin noise, Simplex noise is also a gradient noise which produces very similar results to that of classic Perlin noise, if not slightly denser in appearance. This, in many ways, makes Simplex noise a potential alternative to Perlin noise rather than a straight replacement as the results produced by Perlin noise may be more favourable in some situations where as, the results produced by Simplex noise may be more favourable in others depending on the effect the developer wants to create.

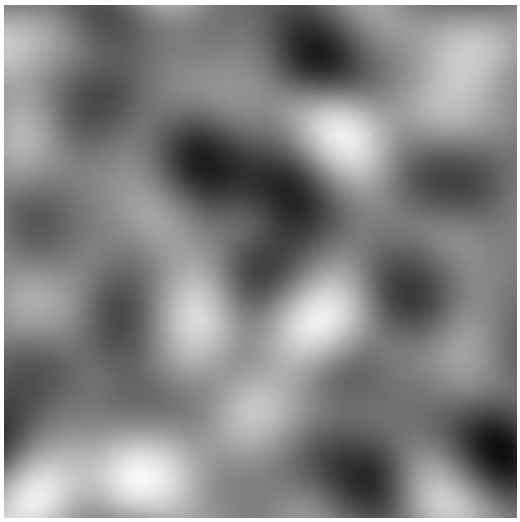


Figure 4. Simplex Noise [8]

2.2.3 Fractal Noise

Fractal noise, sometimes called Fractional Brownian Motion, is the process of adding several octaves of a particular noise function together. In terms of terrain generation, this makes terrain appear more detailed by creating a rough, erosion-like effect on the terrain.

This is achieved by recursively layering one or more layers of a noise function at increasingly higher frequencies which is, of course, computationally much more expensive than simply just using a noise function. However, for procedural terrain generation something like fractal noise is, in many cases, the most effective way to create detailed, natural looking terrain.

"*Fractional Brownian Motion (fBm) is the summation of successive octaves of coherent noise, each with higher frequency and lower amplitude.*" [9]

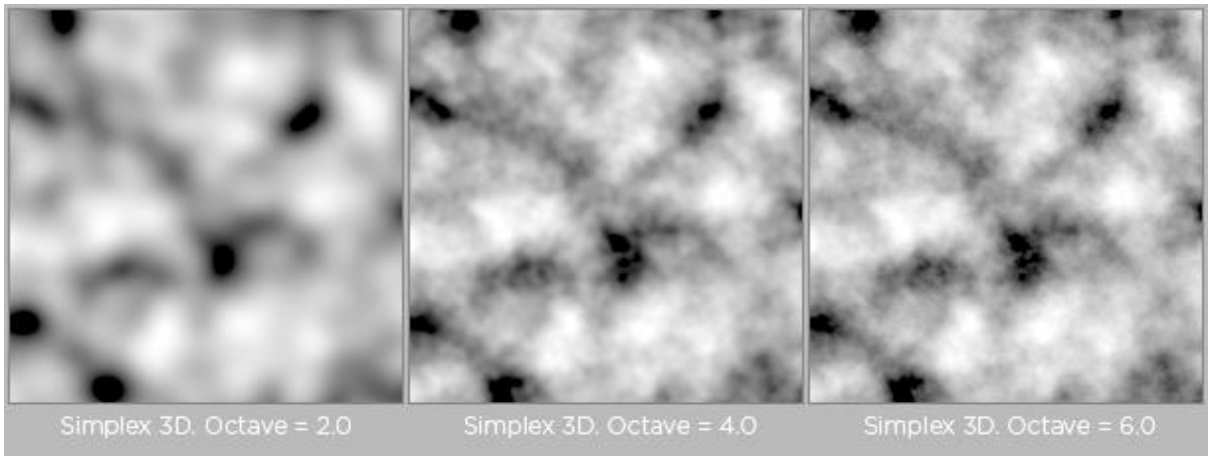


Figure 5. Fractal Noise using 3D Simplex Noise with increasing number of octaves. [10]

2.3 Map Generation

Map generation algorithms are used to generate height maps. Height maps are a way of storing height values in a file for use with computer graphics. These can be in the form of an image such as a bitmap but can also be stored in text files. It is also possible to generate heightmaps using other methods such as noise algorithms.

"*A heightmap or heightfield is a raster image used to store values, such as surface elevation data, for display in 3D computer graphics.*" [11]

These heightmaps are then used to map the X and Z position of a vertex to a height value. For example, the X and Z values of a vertex will map to a corresponding position on the heightmap which will determine the Y height value of the coordinate.

Heightmaps are useful for terrain generation in video games where performance is crucial as they can be quickly and easily generated. In addition, they are quick to read from and don't require too much space in memory as their file size a fairly small.

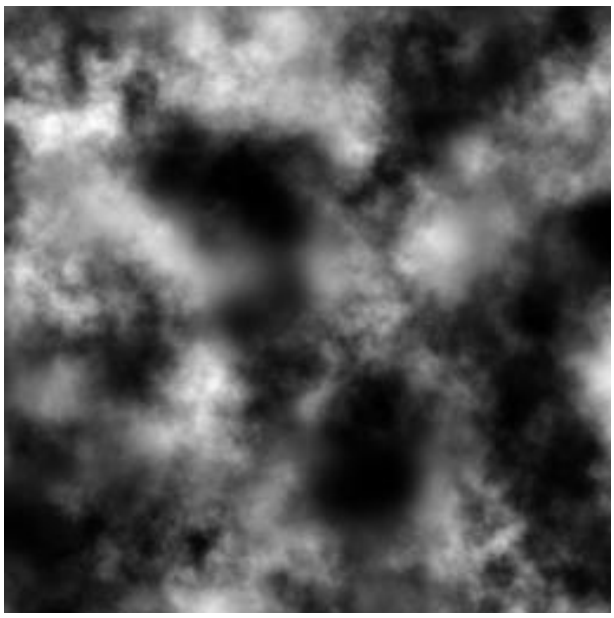


Figure 6. Example Heightmap [12]

2.3.1 Diamond-Square Algorithm

To understand the Diamond-Square algorithm it is first useful to understand the Midpoint Displacement algorithm.

Midpoint Displacement is the older, arguably inferior version of the Diamond-Square algorithm. It has almost been entirely replaced by Diamond-Square as even when searching for the Midpoint Displacement algorithm many results return information regarding Diamond-Square. These algorithms are very similar. However, Midpoint Displacement is carried out horizontally and vertically while Diamond-Square is, for the most part, calculated using diagonals, hence the name Diamond-Square.

"*The Midpoint Displacement Algorithm is a method of generating a height map by assigning a height value to each of the four corners of a rectangle, and then subdividing the rectangle and each resulting child into four smaller rectangles.*" [13]

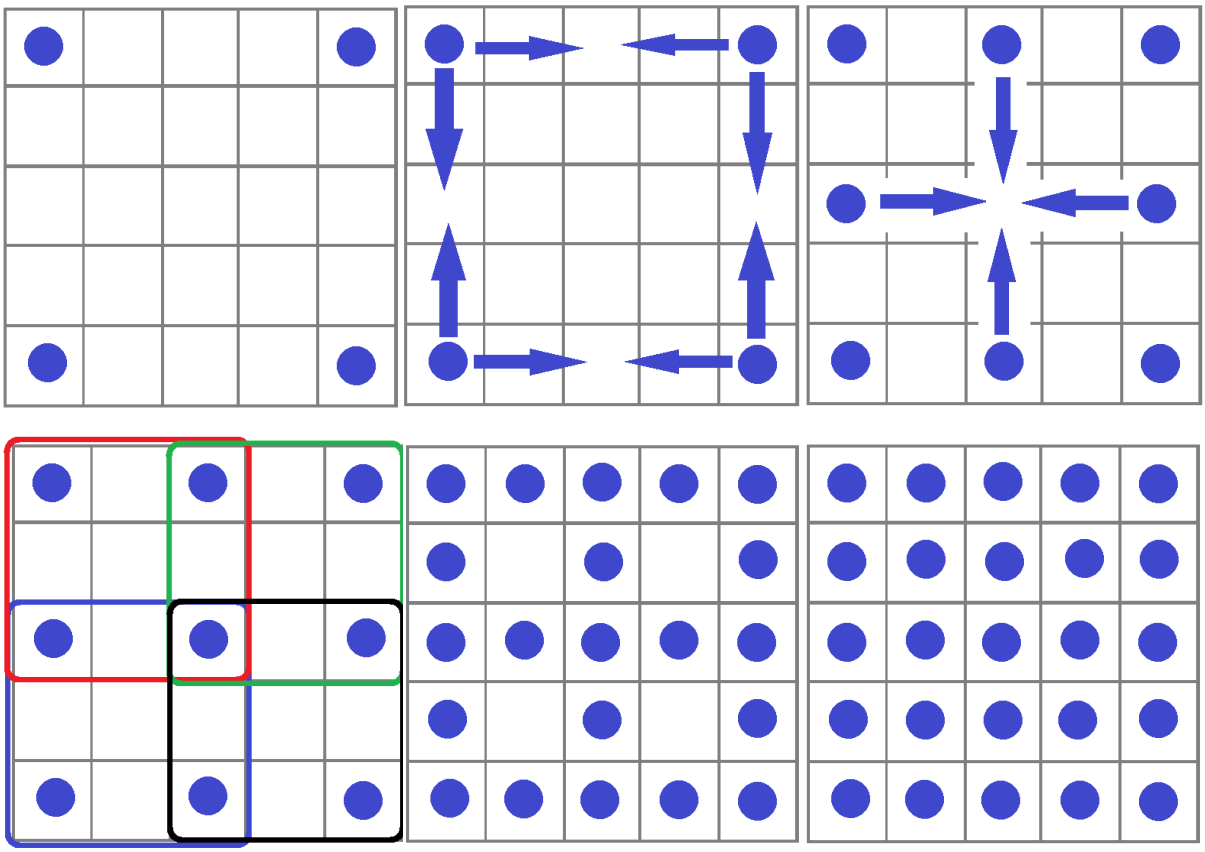


Figure 7. Visual representation of Midpoint Displacement

It can be seen in Figure 7 above that Midpoint Displacement begins with the outer corner points and calculates the points at the centre of each outer column and row before calculating the centre of the newly calculated points. The shape is then divided into four before this method is then repeated until all points for the height map have been generated. A common problem with this method is that it can create visual artefacts which is something that Diamond-Square aims to solve.

Diamond-Square is considered a direct improvement over Midpoint Displacement. It aims to reduce the visual artefacts created when using Midpoint Displacement by calculating points using a diamond step which works diagonally and a square step which works vertically and horizontally.

"*The Diamond-Square Algorithm is an improvement on the Midpoint Displacement Algorithm which reduces the visual artefacts of the algorithm by including an intermediate step which considers diamond shaped squares.*" [14]

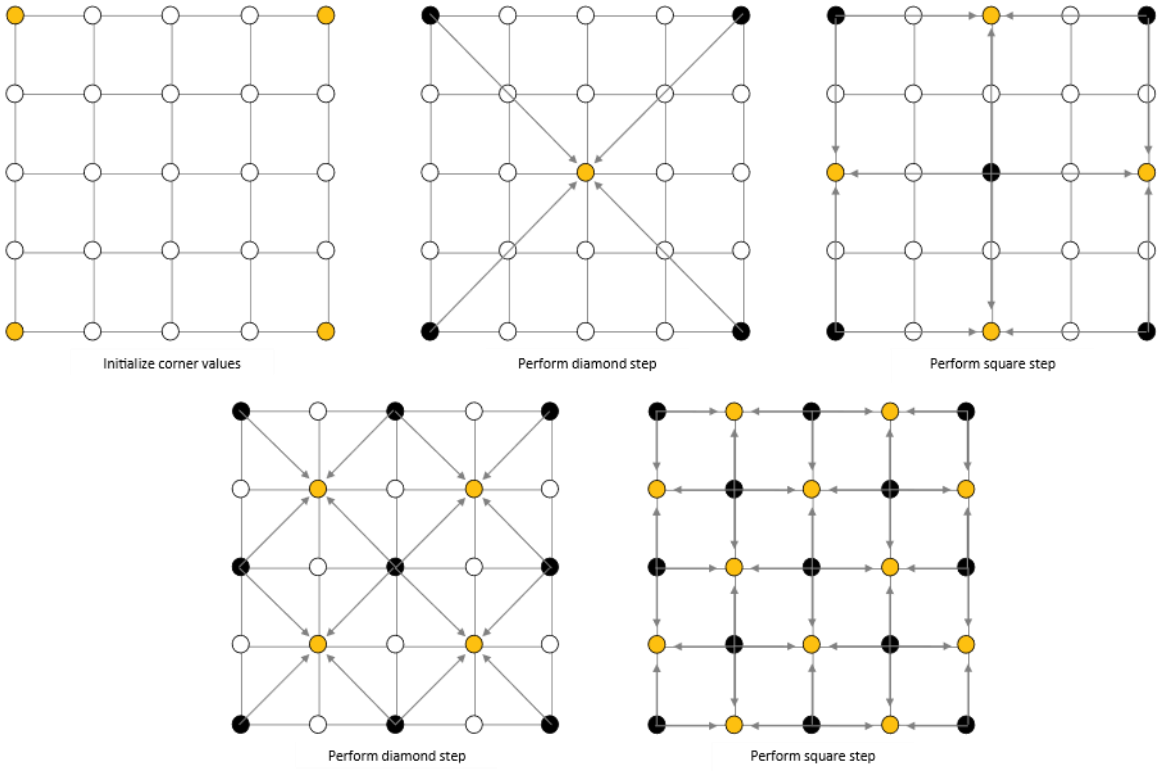


Figure 8. Visual representation of Diamond-Square [15]

As seen in Figure 8 Diamond-Square generates points in diamond shaped patterns which works to reduce the visual artefacts created when using Midpoint Displacement. However, Diamond-Square is not yet perfect as the terrain that is generated suffers from axis aligned ridges [14]. Even with this problem Diamond-Square is a good example of an algorithm commonly used for the generation of heightmaps for video game terrain generation and is worth comparing against the previously discussed noise functions.

Chapter 3. Design and Implementation

3.1 Software and Technologies

There are a variety of software and technologies needed to carry out carry out this type of project. These include a programming language, some type of graphical Application Programming Interface (API) and an Integrated Development Environment (IDE). There are a great number of options for each of these however, choosing the most appropriate is essential as they drive the entire implementation of this dissertation project.

3.1.1 Programming Language: C++

The programming language of choice is C++.

C++ is a language dating back to 1979 when *Bjarne Stroustrup* [16] began to add object-oriented Programming (OOP) into the C language which he called "C with classes". In 1983 this name was changed to C++ which contained many of the low-level functionalities of C but in addition, many high-level features like OOP. [17]

C++ is highly regarded within video game development due to the amount of control it offers the developer. This is because it offers low-level as well as high-level functionality which allows for lightning fast performance as long as the developer understands how to correctly use the language and optimise their code. This makes C++ a perfect choice for this project as the speed at which code is executed is very important so that the performance data collected during testing is directly related to the performance of the terrain generation techniques as much as possible.

"*C++ compiles directly to a machine's native code, allowing it to be one of the fastest languages in the world*" [18]

C++ is not perfect though. Even though it is extremely fast this comes at a cost as it is one of the more difficult languages to use correctly. This is because C++ is very flexible therefore, code can be compiled and executed even if it is brimming with bugs. For example, memory should be handled manually in C++. If the programmer continues to create objects without freeing the memory used by previous objects, the system will eventually begin to run out of memory which can cause many problems such as huge drops in performance and crashing.

3.1.2 OpenGL

OpenGL (Open Graphics Library) is a graphics API which simply enables the rendering of graphics onto screen and is commonly used for rendering 2D and 3D graphics for software such as video games. It is an extremely popular API due to the portability it offers the developer in that OpenGL applications can display consistent results on any compatible hardware no matter the platform.

"*OpenGL has become the industry's most widely used and supported 2D and 3D graphics application programming interface (API), bringing thousands of applications to a wide variety of computer platforms.*" [19]

OpenGL by itself is actually a specification designed by *Khronos Group* [20]. This specification details what each function should do and how it should output and is then implemented by, in most cases, the manufacturer of the graphics hardware through the drivers they release.

OpenGL is the graphics API of choice as it is perfect for this project. Graphics should render consistently regardless of the hardware used and the nature of the project does not ask for more than simply rendering the graphics to screen.

3.1.3 NCL Codebase

The NCL Codebase is a codebase provided by the *Newcastle University School of Computing* [21] for students to use when learning and developing video game graphical content. It includes a long list of useful functionalities with some of the most important being Vector 2, Vector 3 and Vector 4 classes for use with for example, texture coordinates, vertex positions and vertex colour attributes respectively. Included is also support for matrix manipulation, a moveable camera, timer, renderer, shaders and many more.

This is an invaluable codebase for this project as it provides all of the lower level functionality which would otherwise need to be implemented before development of the project could begin.

3.1.4 Visual Studio 2017

Visual Studio is an integrated development environment (IDE) created by Microsoft [22]. It is very commonly used among the gaming industry with C++ as it offers a clean and intuitive user interface while also supporting over 4000 extensions [22] in the rare case that a user needs some extra functionality not already included with Visual Studio.

"*Best-in-class tools for any developer*" [22]

Visual Studio 2017 is a perfect choice for a project like this as it has all the functionality required with the added bonus that the NCL Codebase used for this project was also created in and for use with Visual Studio, so the codebase works without any issues.

3.1.5 Fraps

A benchmarking tool is needed to record the performance for each terrain generation technique during the testing phase of development. Fraps is perfect as it has built in functionality that allows the user to create a benchmark for almost any graphical application by specifying a hotkey to start the benchmark as well as the duration of the benchmark. By

doing this Fraps will monitor and calculate the minimum, maximum and average FPS for the benchmarked application once completed.

"*Show how many Frames Per Second (FPS) you are getting in a corner of your screen. Perform custom benchmarks and measure the frame rate between any two points. Save the statistics out to disk and use them for your own reviews and applications.*" [23]

3.1.6 GPU-Z

"*GPU-Z is a lightweight system utility designed to provide vital information about your video card and graphics processor.*" [24]

GPU-Z will allow every aspect of the graphics card to be monitored throughout the benchmark process of each generation technique so that not only can the performance between each technique be compared, but also the effect that they have on the system hardware. GPU-Z is very suited to this as it can represent the readings of the graphics card sensors on a graph so that any erroneous changes throughout the benchmark process will be easily noticed.

3.1.7 HWMonitor

HWMonitor is similar to GPU-Z in that it is monitoring utility software. However, it also provides monitoring for almost every major piece of hardware within the PC. This is useful for monitoring Central Processing Unit (CPU) behaviour throughout the duration of the benchmark.

"*HWMonitor is a hardware monitoring program that reads PC systems main health sensors: voltages, temperatures, fans speed.*" [25]

3.2 Implementation

3.2.1 The Project

Before the implementation of the project could commence. It would need to be decided how the project should be managed. Two of the most common forms of project management in computing are the Waterfall method and the Agile method however, many are now moving towards agile due to the extra flexibility in the ability to make changes during development. Because of this, it was decided that an agile methodology would be used throughout this project.

The next step before implementation was to reiterate the project goals to ensure the focus of the project would be maintained throughout. In addition, a plan for the order of implementation for the various functionalities needed for the project would need to be set up to ensure it was clear what was currently being implemented and what was due to implemented next.

The implementation sections below describe the reasoning for the chosen project management method, reiterate the intended project goals and describe the implementation of the project in the order in which each stage was carried out.

3.2.1.1 Agile Methodology

Throughout the project an agile management method was used. This fit the nature of the project as a meeting was held every two weeks. This meeting was held with a selection of four Newcastle School of Computing staff members with a vast amount of knowledge on the subject of the project. The purpose of this meeting was to measure the progress between each meeting and then decide on the best way to spend the upcoming two weeks. Because of this, an agile method was the obvious choice.

The Agile method of project management is a method very commonly used in software development projects as it helps reduce the harmful effects of the unpredictability of building software by encouraging an iterative approach to the development. This is a better way to manage this type of project than for example, the traditional waterfall method as this method places a lot of restrictions on change throughout the project and one stage must be completed before the next stage can begin while agile offers much more flexibility.

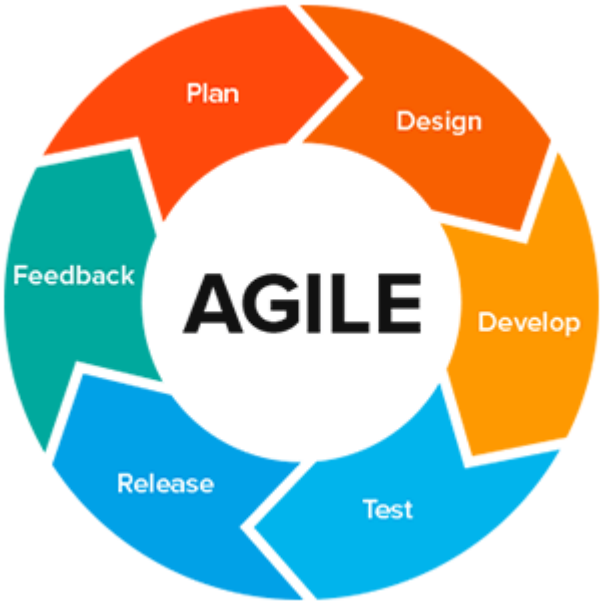


Figure 9. Agile Management Method [26]

Figure 9 shows the basic flow of the agile method. So small iterations to the design and development of the project are made in between the feedback stage. In this case, the feedback stage being the fortnightly meetings which directly drive the design and development for the next two weeks. This is vital as it allows the project space for possible changes to development. If, for example, some functionality is implemented differently to how expected, it will be possible to alter this without too big a change to the overall project. In addition, the meetings force progress of the project which will help ensure it is completed within the time frame.

3.2.1.2 Project Goals

The goals in terms of the implementation of the project are to implement a suitable method that generates a mesh capable for use with terrain generation techniques. The mesh would need to be quick to generate so that the differences between each algorithm can be measured during testing as accurately and precisely as possible. The mesh would also need to have the ability to scale to a variety of sizes for possible testing purposes.

The second main goal of the implementation stage was to get the algorithms that are to be tested, implemented. The implementations of the techniques should execute as quickly as possible and should produce good quality results when applied to the implemented project mesh code.

The plan for the implementation of the above goals was to get the base mesh up and running as quickly as possible to ensure a good amount of time was left to get some good implementations of the terrain generation techniques implemented and to begin to play around with them to ensure they work correctly with the already implemented mesh. If this

was not the case, then there would be enough time to spend altering the mesh code so that they all work as intended.

3.2.2 Base Mesh

In order to apply the procedural terrain generation techniques a way to generate a suitable base mesh was needed that expands across the X and Z axis so that the techniques can generate the height values for the Y axis. The speed and scalability of this mesh was extremely important so that the difference between each terrain generation technique could be measured as accurately and precisely as possible.

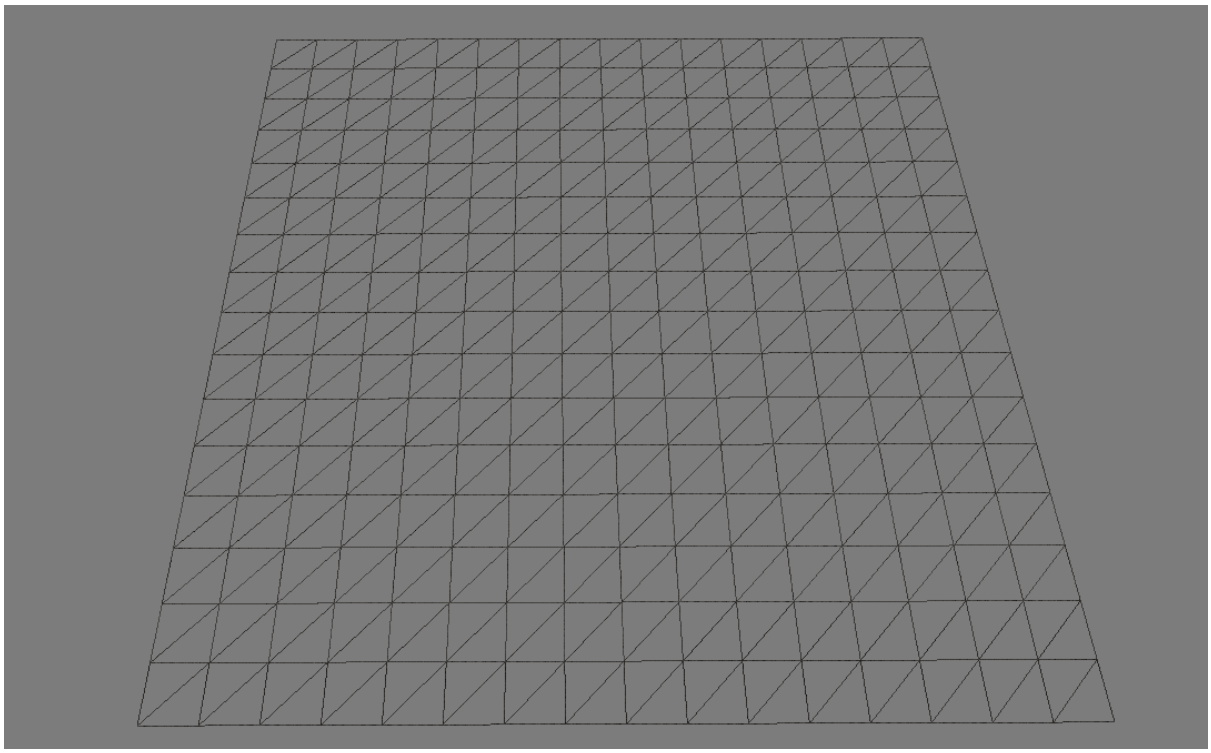


Figure 10. Generated Base Mesh for Project

Figure 10 above shows a 16x16 (X, Z) base mesh generated with Y height values of 0. Once this mesh is generated with some Y values provided by the terrain generation techniques it will begin to look a lot more like terrain. The implementation of the mesh was carried out using vertices and indices allowing for it to be one single triangle strip which is one of the rendering primitives available within OpenGL.

3.2.2.1 Triangle Strips

GL_TRIANGLE_STRIP or triangle strips are one of the OpenGL rendering primitive types and are the primitive of choice for creating the mesh required for this project. They are useful as they allow many triangles to be created with a reduced number of vertices compared to using just the triangle primitive.

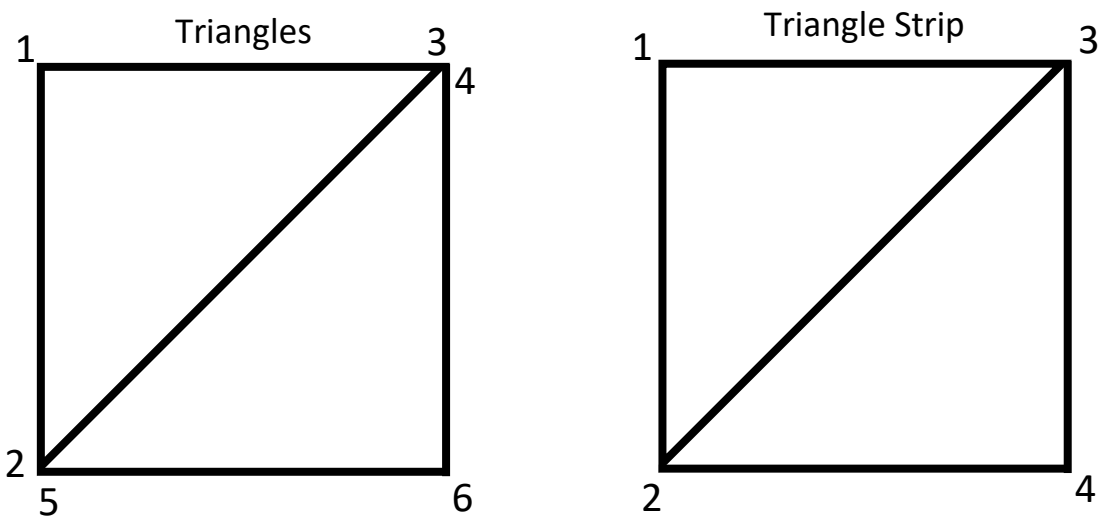


Figure 11. Comparison of vertex count for Triangles and Triangle Strip

Figure 11 shows how the triangle strip is much more efficient to use than just triangles. With regular triangles you must create three new vertices for each triangle even if some of the triangles use a repeated vertex. This is inefficient as it can result in many duplicate vertices. Triangle strips on the other hand, allow a new triangle to be drawn using the last three previously specified vertices. For example, in Figure 11, vertices 1, 2 and 3 result in the first triangle. From here adding vertex 4 then creates the second triangle as the triangle strip knows to use the last two previous vertices, vertices 2, 3 and the newly added 4. This means only four vertices are needed to draw two triangles compared to needing six vertices using the normal triangle primitive. This is extremely beneficial because if for example, the mesh that is being created is made up of many thousands of triangles then triangle strips remove the need for duplicate vertices which would save a huge amount of memory.

There are of course other rendering primitives available in OpenGL such as quads, triangle fans, lines and several more. However, these all bring with them their own issues. For example, if quads were to be used, they consist of four vertices so the vertices of a quad can exist on more than one plane which creates problems such as when calculating surface normals as there would be two sets of surface normals rather than one. This is not a problem when using triangles as all of the points of a triangle always exists across one single plane. Because of this, triangle strips are absolutely the most suitable rendering primitive to use for this project.

3.2.2.2 Vertices and Indexing

The use of the triangle strip primitive type makes it beneficial to understand how to best utilise vertices and indices.

When creating a polygonal mesh, vertices are used to act as points which are then connected together by edges which are basically just lines. When using a 3-Dimensional coordinate space, which is necessary for this project, vertices are made up of an x, y and z position value however, they may also be given a colour attribute, texture coordinate and

normal so for a single vertex, quite a lot of data may need to be stored. Throughout the creation of a mesh, if the mesh uses more than just a few triangles, many vertices will usually need to be repeated which means lots of vertex data will be duplicated. If, for example, a mesh contains thousands or even millions of triangles, it is likely that a huge amount of vertex data would need to be duplicated which could take up an unnecessarily large amount of memory. It is possible to reduce this memory cost using a method known as indexing.

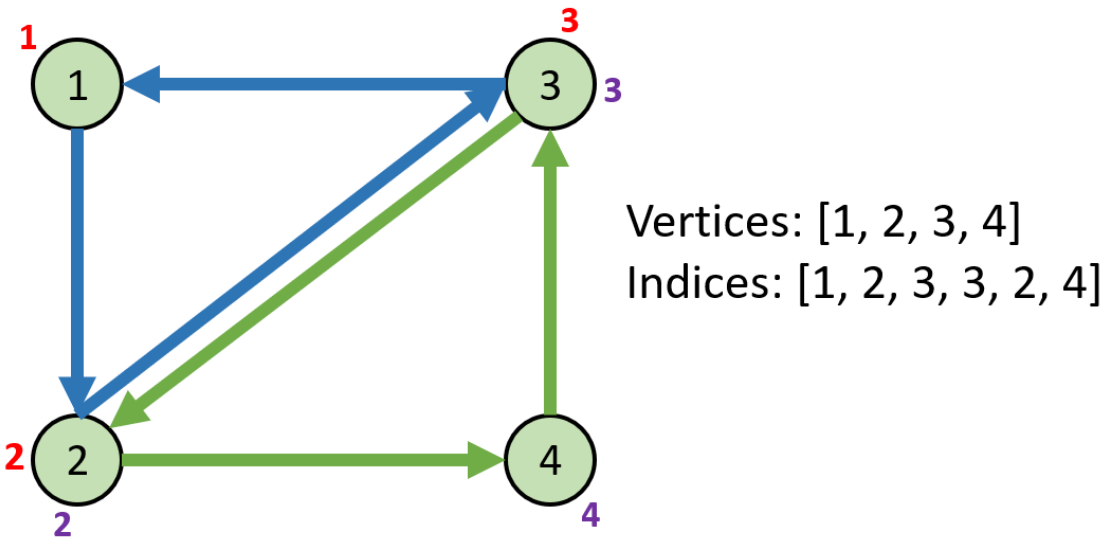


Figure 12. Indexing

Indexing allows for each vertex to only be stored once for example, in an array. While a second set of usually unsigned integer values are used to effectively connect each of the vertices by specifying the order in which each vertex should be connected using the index positions of the vertex in the array. Unsigned integer values are used as they do not allow negative values which prevents problems like trying to index into a negative value of an array as it is impossible to have a negative number of vertices. This removes a lot of the memory problems introduced with duplicating vertices since now, only an unsigned integer will be duplicated in comparison to an x, y and z position, colour attribute, texture coordinate and normal.

Figure 12 gives a visual example of indexing. Only four vertices are being used to draw two triangles using the OpenGL GL_TRIANGLES rendering primitive where every three vertices define a new triangle. Rather than duplicating vertices 2 and 3, an array of indices containing the order of the vertices is defined which instead only duplicates two unsigned integers which is usually much more efficient than duplicating entire vertices.

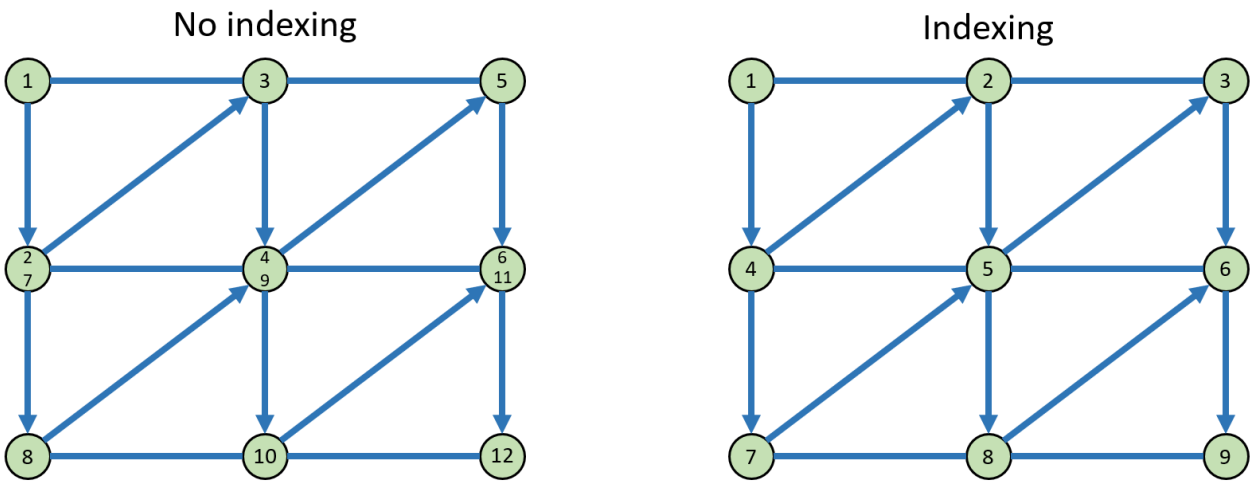


Figure 13. Vertices without indexing compared to with Indexing

Figure 13 shows the difference in the number of vertices needed when not using indexing compared to when using indexing on a very small scale. Without indexing twelve vertices are needed to complete the 3x3 grid while only nine vertices are needed when using indexing. This is because without indexing every vertex not in the first and last row needs to be duplicated which, in this case, is three vertices that need to be duplicated. However, when using indexing it requires no duplicate vertices. This may not seem like too much of a saving in memory from this example, but on a much larger scale such as a 256x256 size grid 65,024 (256*254) vertices would need to be duplicated which is a lot of data to store twice especially if they all require the vertex attributes described above to be duplicated too.

Indexing can have a negative impact in some cases. It is extremely rare but if a defined mesh has no duplicate vertices then creating an array of unsigned integers for indices will use up more memory than simply just using the original vertices. So, in this case it would be beneficial to not use indexing but as said above, this case is very rare.

3.2.2.3 Winding

When creating a mesh, it is important to consider the order of the vertices or, if using indices consider the order of the indices to ensure they follow the correct winding order. This is important because it determines which polygons are facing forwards and backwards. For example, OpenGL considers vertices that are ordered in a counter clockwise manner to be front facing while clockwise ordered vertices are back facing. Knowledge of the direction a polygon is facing is extremely important as it can cause problems with the direction of for example, normals for lighting. In addition, it can sometimes be beneficial to cull polygons that are back facing as they are usually not being seen therefore, culling them can increase the performance of the simulation. This shows the importance of winding order because if the order is not correctly maintained throughout a particular mesh, the simulation could incorrectly cull vertices that should be visible but are labelled as back facing due to an incorrect winding order.

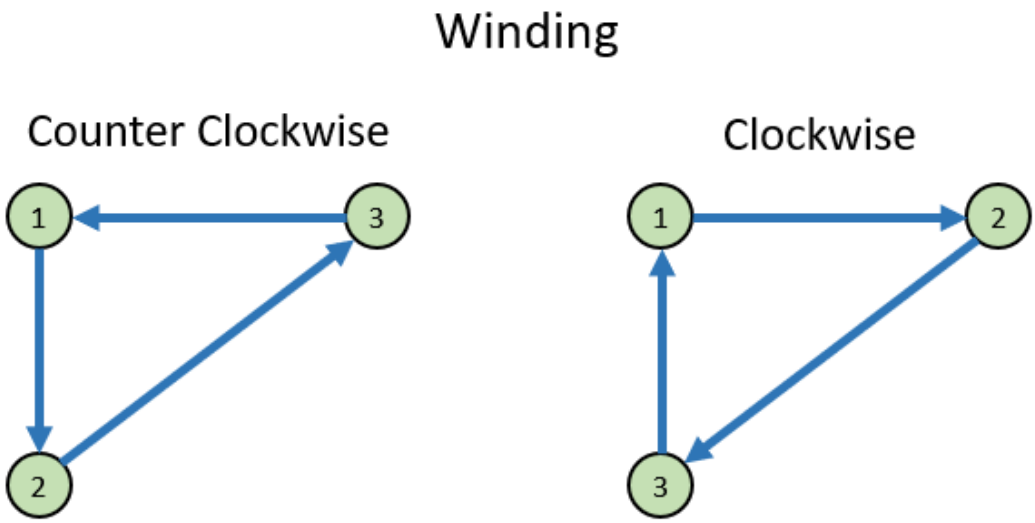


Figure 14. Counter Clockwise compared with Clockwise Winding

Figure 14 shows an example of two triangles, one drawn using a counter clockwise winding order and the other drawn using a clockwise winding order. The order of each of the vertices in both examples differ slightly to achieve this. So, if using OpenGL, the triangle on the left would be considered front facing while the triangle on the right would be considered back facing. If the simulation was for example, setup to cull backwards facing polygons then the right triangle would not be seen even though realistically, it probably should be visible. This shows the importance of ensuring the winding order of vertices is maintained correctly.

"*When vertices are broken down into Primitives during Primitive Assembly, the order of the vertices relative to the others in the primitive is noted. The order of the vertices in a triangle, when combined with their visual orientation, can be used to determine whether the triangle is being seen from the 'front' or the 'back' side.*" [27]

3.2.2.4 Degenerate Triangles

When using indexing for a grid-based mesh it offers a way to simply and easily generate each of the vertices of the grid without yet worrying about the order they are generated.

```
for z = 0 to gridDepth do
    for x = 0 to gridWidth do
        add vertex(x, z)
    end
end
```

Figure 15. Pseudocode for Generating Mesh Vertices

Figure 15 shows the basic method used in this project to create a grid of vertices that spans across the X and Z directions. When it comes to adding indexing to this grid of vertices it is important to keep the winding order in which the vertices are connected consistent throughout the mesh, but it is also important to ensure each vertex is connected as intended. In this case, there are two ways in which the vertices can be connected.

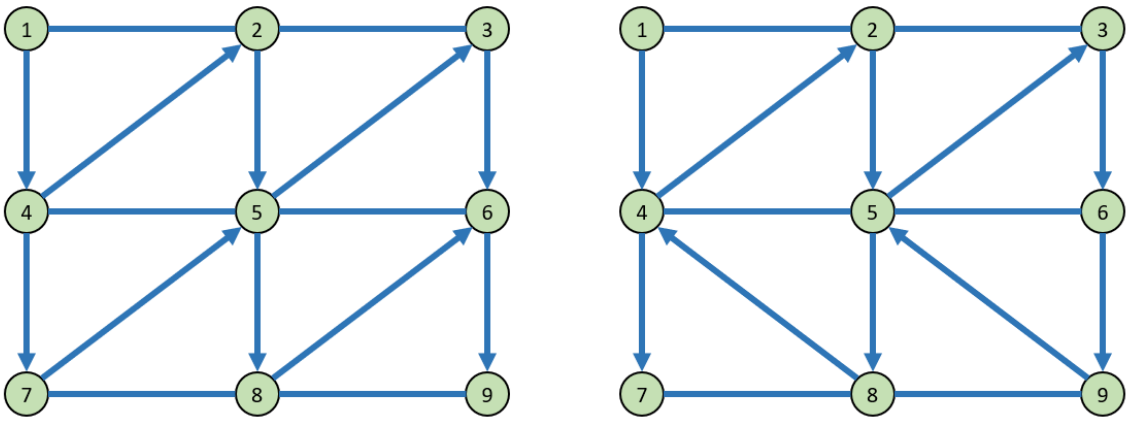


Figure 16. Consistent compared against Alternating Rows

Figure 16 shows the two ways in which the indexing for this can be tackled. Either all of the rows are created in the same direction, from left to right. Or the rows alternate in something similar to a zig-zag pattern. The former is the most difficult of the two to implement but the latter can cause issues with winding and setting the correct texture coordinates as the rows alternate each time therefore, the first implementation is being used in this project.

For this to work, vertex 4 needs to begin the second row while the first row ends at vertex 6 so a method known as degenerate triangles are used. Degenerate triangles are simply triangles defined by three collinear points. These are points which all lie on the same straight line. This means the triangle is simply a line and has no area which, in the case of OpenGL, causes the triangle to not be drawn to screen at all. Therefore, allowing the next row of vertices to begin at vertex 4 rather than 6. One exception, however, is viewing the mesh as a wireframe. In this case the connections of the degenerate triangles will be shown but realistically, this will only be viewable by a developer.

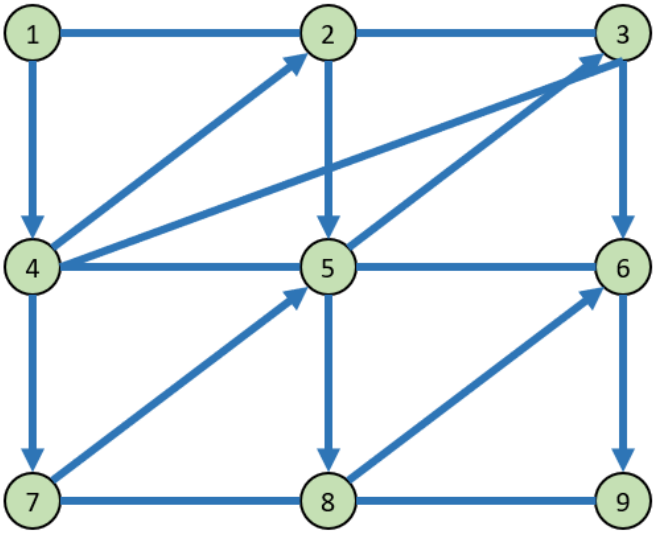


Figure 17. Example grid without using degenerate triangles

Figure 17 shows what would happen to the mesh if degenerate triangles were not used. Because of the nature of the triangle strip rendering primitive, the previous two vertices are used to create the next triangle so when beginning the second row the last two vertices are vertex 3 and vertex 6. So, when specifying an index for vertex 4, a triangle is created using vertices 3, 4 and 6 which is not the expected output for this project.

Implementing degenerate triangles is very simple in the case of this particular grid-based mesh. To do this the indices for the last vertex and first vertex in each row are repeated.

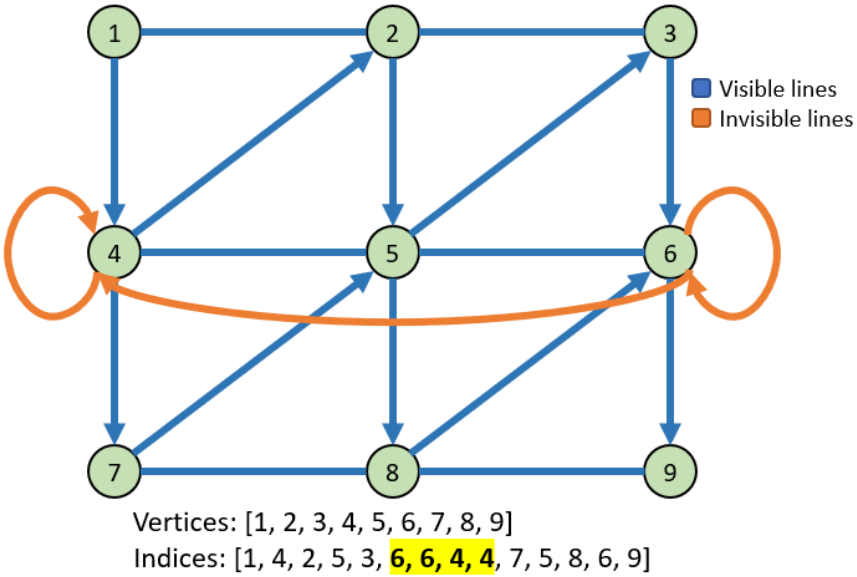


Figure 18. Visual implementation of a Degenerate Triangle

Figure 18 shows how degenerate triangles are implemented. Here, vertex 6 and vertex 4 are both repeated. The degenerate triangle exists along the line using indices 6, 6, 4 as these are collinear and therefore, have no area and will not be drawn. Vertex 4 is then repeated so that a triangle would not be created between vertices 6, 4 and 7 but in addition, it also keeps the winding order of the vertices consistent. This method is repeated at the end of each row of vertices until all vertices are connected.

3.2.3 Algorithms

There are four algorithms that were decided on being used for this project which are Perlin Noise, Simplex Noise, Fractal Noise and the Diamond-Square algorithm. All of these algorithms are, or were at some point, very heavily used for procedurally generating terrain for real time simulations. Which makes these algorithms good subjects for testing with the goal of trying to find which is the all-around best for terrain generation.

3.2.3.1 Perlin Noise

The implementation for the Perlin Noise function used for this project was found on a website while researching Perlin Noise [28]. It is a direct conversion of a Ken Perlin Java implementation to a C++ implementation of improved Perlin Noise which Ken Perlin produced in 2002 as a direct improvement to his classical noise however, the core logic of the algorithm remains the same with only a few changes.

"*Two deficiencies in the original Noise algorithm are corrected: second order interpolation discontinuity and unoptimal gradient computation. With these defects corrected, Noise both looks better and runs faster.*" [29]

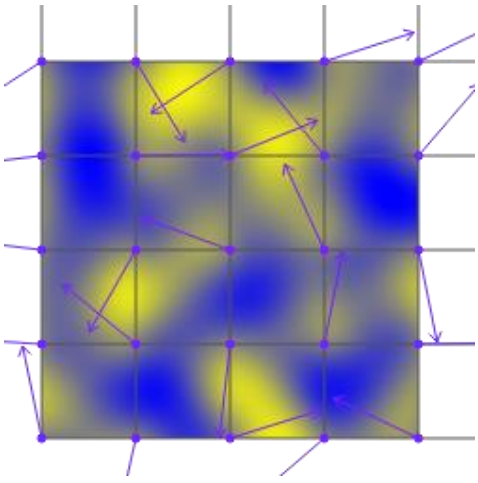


Figure 19. Ken Perlins' Improved Noise with Visualisation of Gradient Vectors [30]

This implementation remains largely the same as Ken Perlins' original implementation of the algorithm and overall, is not too complex. The algorithm aims to create noise by generating a series of pseudo-random gradient vectors based on the values input into the function. Pseudo-random meaning that if the same values were to be input several times, the result value would be the same result each time. This may not be great in terms of complete

randomness, but it does allow the result of the algorithm to be determined and controlled which may be useful depending on the implementation. For this implementation, this does not matter all that much.

Using this implementation is extremely simple. Once the code is in Visual Studio it is simply a matter of calling the function by inputting the X and Z position values as parameters to then receive a value bound between -1 and 1 which can be scaled up and used as an appropriate Y height value. If the bound Y value is not scaled up, it is almost impossible to notice a change in height which is why the scaling is necessary. However, scaling too much results in extremely pointy looking terrain so it is important to scale the value to a suitable amount.

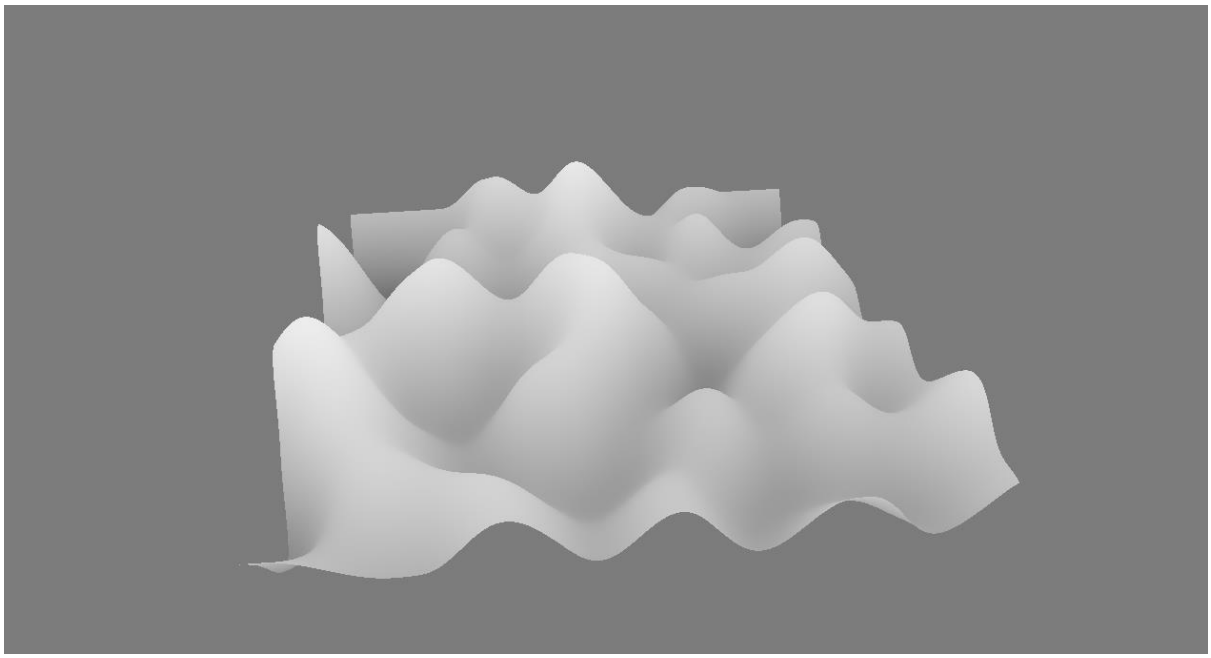


Figure 20. 512x512 of Perlin Noise Generated by the Project Code

3.2.3.2 Simplex Noise

The implementation for Simplex Noise was found on GitHub. This implementation is based on the 2001 Simplex Noise implementation outlined by Ken Perlin [31]. It includes a 1D, 2D and 3D version of Simplex Noise which was perfect for the purpose of this project. Like Perlin Noise before it, Simplex Noise is a pseudo-random type of noise generation meaning the same input will always result in the same output which may or may not be desirable depending on the implementation. For this implementation however, this does not cause any problems.

"*Perlin's 'Simplex' Noise (2001) rather than placing each input point into a cubic grid, based on the integer parts of its (x, y, z) coordinate values, placed*[sic] *them onto a simplicial grid (think triangles instead of squares, pyramids instead of cubes…)*" [9]

Simplex Noise is usually regarded as a direct replacement to Classic Perlin Noise which Ken Perlin outlined in 1984. This is due to simplex noise being much more efficient to use at

higher dimensions however, the way these noise functions are being used for this project, this may not become too much of a differentiating factor, if at all.

"*A fundamental problem of classic noise is that it involves sequential interpolations along each dimension. Apart from the rapid increase in computational complexity as we move to higher dimensions, it becomes more and more of a problem to compute the analytic derivative of the interpolated function.*" [3]

Once the code was implemented into Visual Studio, like the previous Perlin Noise function, it was as simple as calling the function by inputting the positional X and Z values as parameters for the Simplex Noise function. This then returns a Y value bound between -1 and 1 which can be scaled up appropriately so that the change in height can be recognised but not by too much to avoid a pointy, unnatural looking terrain.

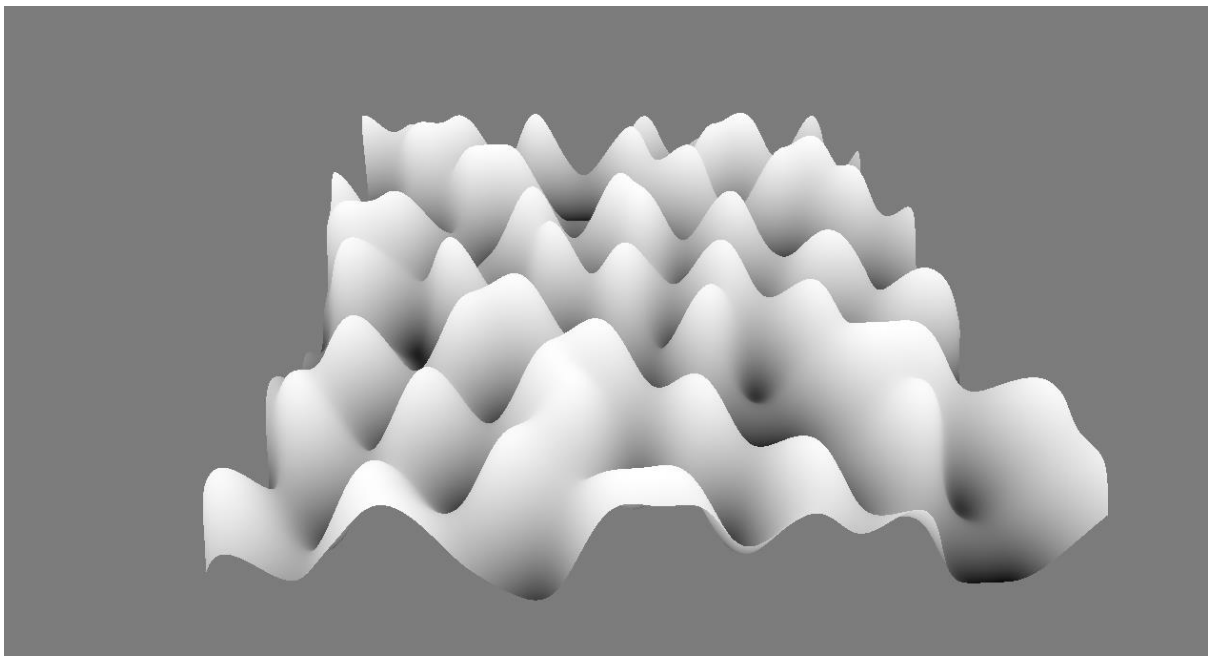


Figure 21. 512x512 of Simplex Noise Generated by the Project Code

3.2.3.3 Fractal Noise

The implementation for Fractal Noise otherwise known as Fractional Brownian Motion was found with the implementation of Simplex Noise [31]. Like Simplex Noise this implementation of Fractal Noise has a 1D, 2D and 3D implementation, more than enough for this project.

Fractal Noise is commonly used to generate more 'detailed' noise than using just a simple noise function such as Simplex Noise or Perlin Noise. The effect offered by this type of noise is similar to that of erosion and can be created by adding together octaves of noise at increasingly higher frequencies and lower amplitudes.

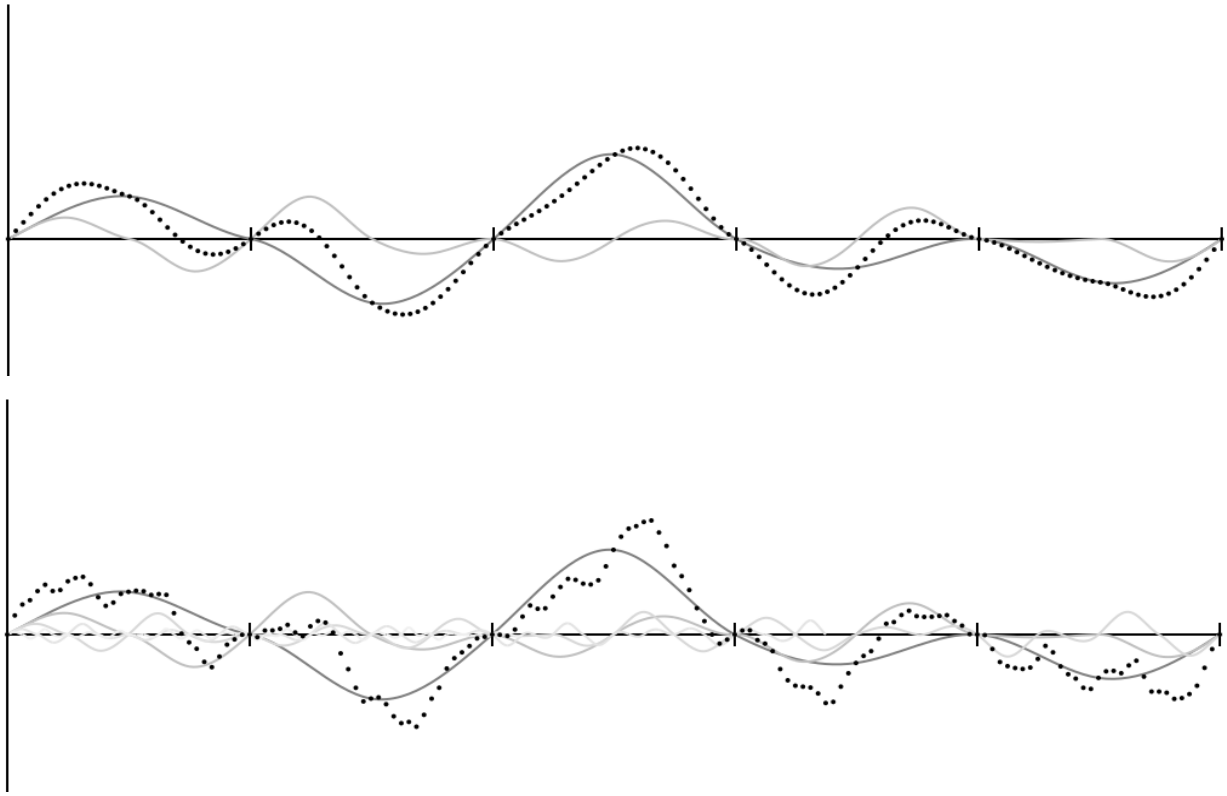


Figure 22. Visualisation of Fractal Noise using Various Perlin Noise Curves [32]

As seen in Figure 22 the light grey curves which represent different frequencies of Perlin Noise can be summed together to create a Fractal Noise which are represented by the dotted curves. This creates much more detail in the curve which is extremely beneficial for procedural terrain generation.

Once the code is implemented into Visual Studio it is simple to run. With some small tweaks the Fractal Noise function can be setup for use with both Simplex Noise and Perlin Noise. The function itself is called using X and Z positions as parameters along with an integer value specifying the number of octaves to use. The higher the number of octaves, the more detailed the generated terrain will appear. This function returns a Y height value which can then be suitably scaled so that the change in height is visible but not so much that it appears unnatural and pointy.

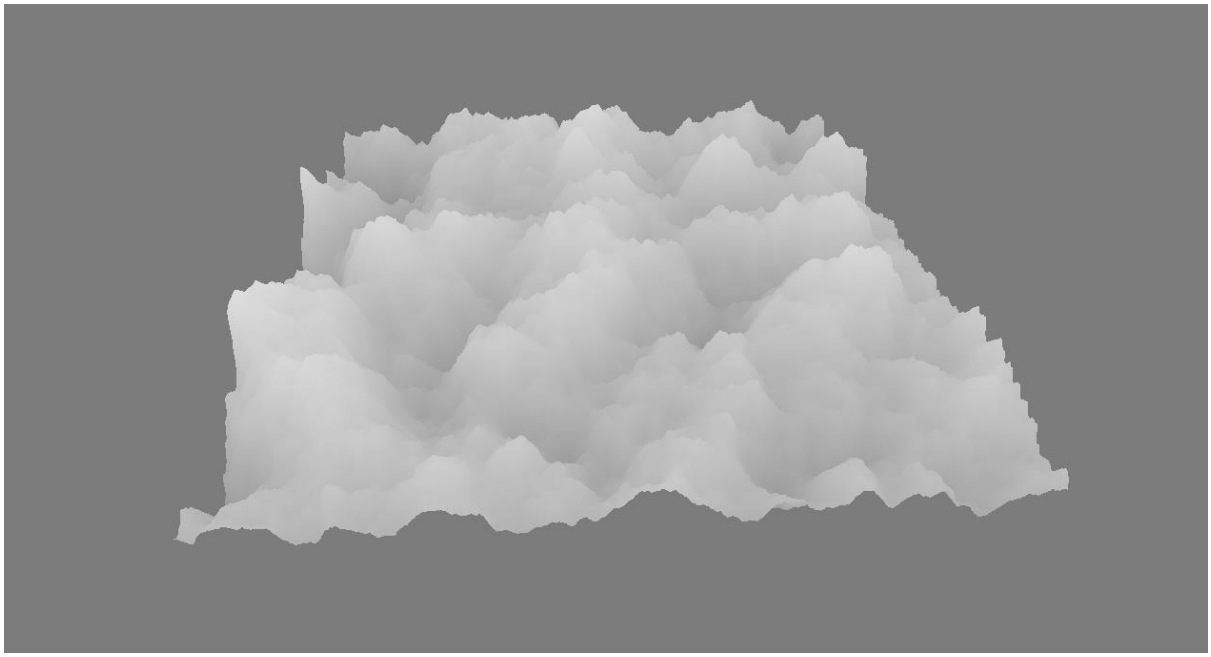


Figure 23. 512x512 of Perlin Fractal Noise 10 Octaves Generated by the Project Code

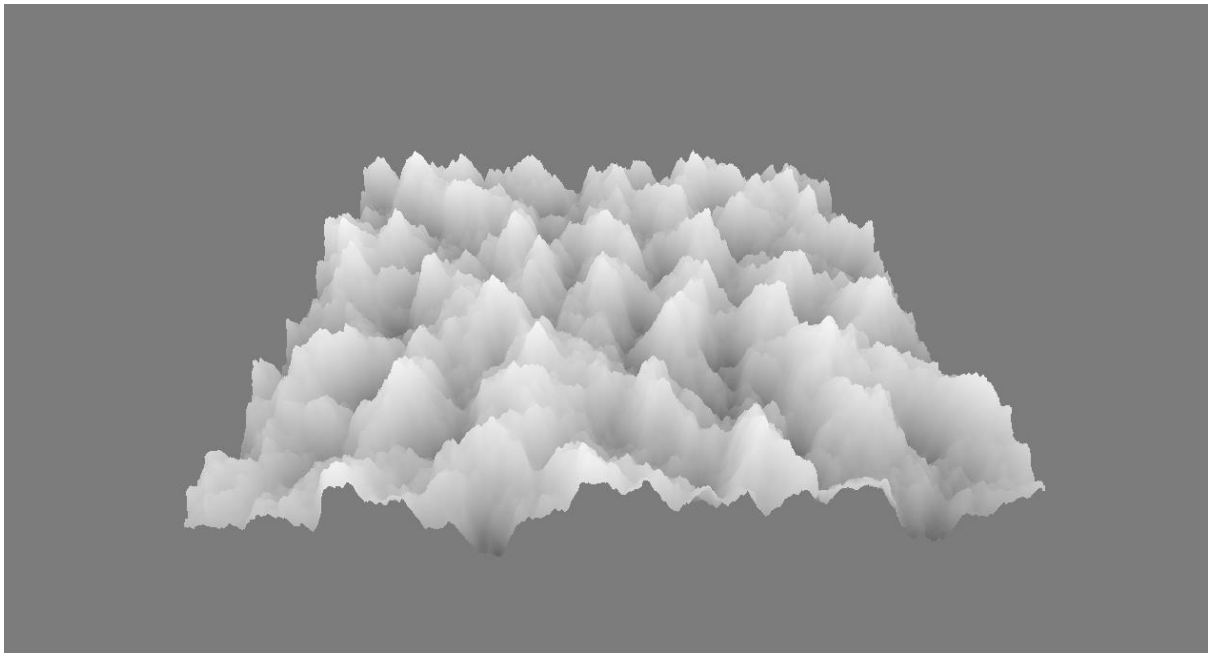


Figure 24. 512x512 of Simplex Fractal Noise 10 Octaves Generated by the Project Code

3.2.3.4 Diamond-Square

The implementation for Diamond-Square was found on GitHub [33]. This implementation was originally intended for use in Unreal Engine however, it works perfectly fine in just OpenGL. Diamond-Square is different from the previous noise functions. Rather than calling a function which generates each specific Y value on the fly, Diamond-Square instead generates an entire heightmap of values. This can then be used to retrieve height values by using X and Z positional values as they each map to a different Y height value.

The implementation of Diamond-Square is based on the idea of filling in a two-dimensional array by calculating values using values that are already calculated. The array must be of size 2^n + 1 so for a terrain of size 128x128 the array must be of size 129x129 which is 2^7 + 1. The algorithm begins by using four corner values for the tile to be generated calculating all of the inside values recursively using a square step and diamond step until all values in the array are filled (see Figure 8).

Using this implementation is extremely simple. The X and Z position values from the base grid can be used to index into the two-dimensional array which will produce a suitable Y height value. This value can then be appropriately scaled up so that the change in height is visible but not so much that it appears unnatural and pointy.

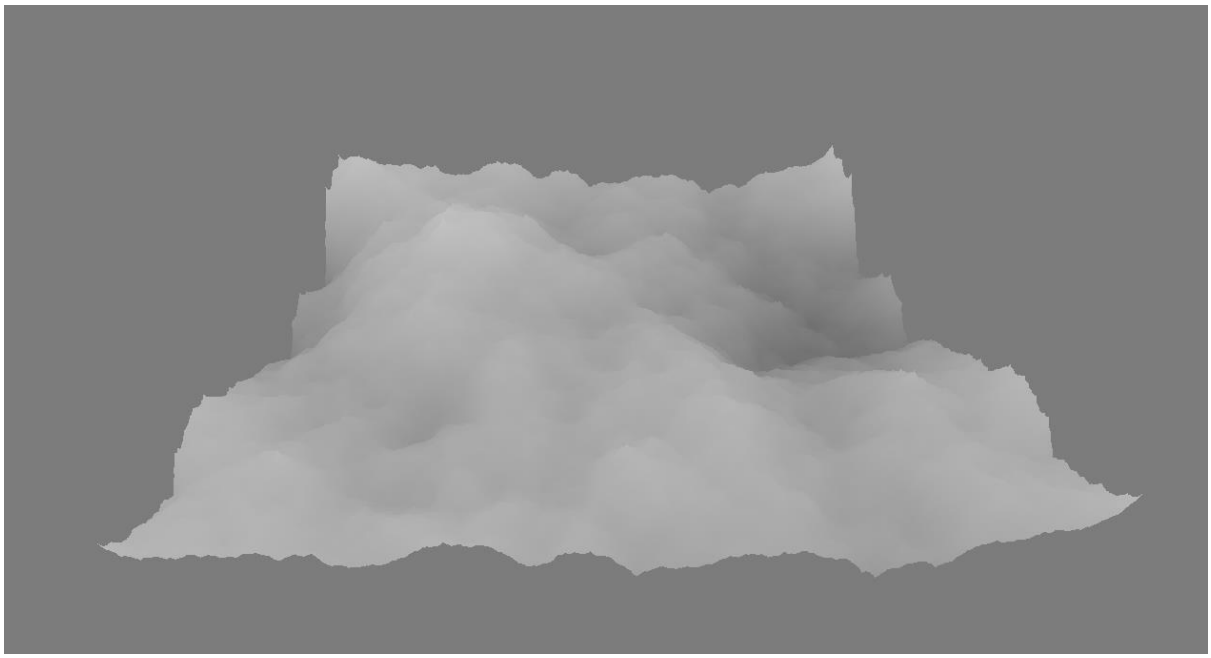


Figure 25. 512x512 of Diamond-Square Generated by the Project Code

3.2.4 Project Output

All of the black and white terrain above was generated by the code created for this project using the implemented algorithms. The intention with the output was to make it look similar to a heightmap. In this case the lower the height value is in local space, the darker it becomes and inversely, the higher the height value in local space, the lighter it becomes. This effect is calculated using a fragment shader. The vertex shader is setup to send the local position of each vertex to the fragment shader which then applies the appropriate colour based on the height.

Chapter 4. Testing and Results Evaluation

4.1 Testing Methodology

The testing phase of the project was extremely important as it would provide the necessary results needed to compare the differences between each implemented algorithm. Because of the nature of this project, testing would need to be carried out as fairly as possible therefore, the only thing that would change between testing each terrain generation technique would be the technique itself.

I order to get a good reading on the performance and speed of each technique a 128x128 sized tile was generated to calculate results based on one tile. Next the scale of tiles was increased to a 25x25 set of 128x128 tiles. This was simply to see if there were any differences between the techniques when they are needed to generate a large amount of terrain at once. In addition to creating an increasing number of tiles of 128x128 the techniques were also tested to generate a single tile of size 512x512 to see how each technique handles the creation of a large single tile.

For each test the CPU, GPU, and RAM usage were monitored throughout. Additionally, and perhaps most importantly, the execution time of each technique was calculated and the minimum, maximum and average framerate were recorded using the FRAPS benchmark utility. In terms of the hardware used for this testing, to get an idea of how the techniques actually performed, the CPU was an Intel i5-8600K at 4.50GHz (GigaHertz), the GPU was a Nvidia GTX 1070 and the RAM used was 16GB (GigaByte) at 4000MHz (MegaHertz).

4.1.2 Test Run

This is an explanation of the basic steps that where taken for each test. These steps remained exactly the same between each test to ensure results were as reliable as possible.

The size and number of tiles would first be specified within the code and the camera was setup to view the maximum amount of the generated tile as possible. Upon running the code, timers were set within the code to calculate the time taken to execute the particular technique being tested. Once everything was loaded the FRAPS benchmarking utility would be started and ran for a total of 30 seconds. This was a suitable amount of time to ensure readings are reliable however, the benchmark does not need to last too long since nothing much happens after the generation of the terrain.

Once the benchmark has completed the results are saved to a Microsoft Excel file and the time taken for the technique to execute within the code will be already conveniently printed in the console. Each test will be tested a total of 5 times to ensure the results are not skewed by any potential outliers.

4.1.3 Base Mesh Performance

Since the mesh is all being generated in code at once it may be important to calculate the speed at which the base mesh without any height values applied runs. This could then be

deducted from the total speed calculated when applying a terrain generation technique to the mesh so that the end result value should be the time taken for the height values of the mesh to be calculated and applied. The problem with this however, is this does not demonstrate the actual performance and time taken to create an entire terrain mesh using each technique so it has been decided that during testing, the total execution time of the mesh will be taken without any deductions of the base mesh speed.

The base mesh performance statistics are still included below for the case of curiosity.

Base Mesh Performance	**128x128**		**512x512**	
Test Number	Value(Seconds)	Value(milliseconds)	Value(Seconds)	Value(milliseconds)
1	0.049	49	0.788	788
2	0.051	51	0.778	778
3	0.05	50	0.78	780
4	0.051	51	0.768	768
5	0.05	50	0.779	779
6	0.051	51	0.765	765
7	0.049	49	0.779	779
8	0.051	51	0.766	766
9	0.049	49	0.789	789
10	0.053	53	0.775	775
Average Time ->	**0.0504**	**50.4**	**0.7767**	**776.7**

Table 1. Base Mesh Performance Results

Table 1 shows the performance values of the execution time for the base mesh. This is the mesh with only values along the X and Z directions and a Y height value of zero across the whole mesh. The performance results above detail the speed to execute a mesh of size 128x128 and 512x512 which was carried out a total of ten times. This is to ensure the speed of the execution of the base mesh is calculated reliably and helps prevent potential outliers from skewing the results. So, from these results it can be concluded that the base mesh is executed in about 50.4 milliseconds at a size of 128x128 and is executed in about 776.6 milliseconds at a size of 512x512.

Looking at the values achieved the speed of execution begins to make sense. The 128x128 sized grid took about 50.4 milliseconds and the 512x512 sized grid took about 776.8 milliseconds. This makes sense since 128x128 = 16384 while 512x512 = 262144. This is a 16 times difference in size and 776.8/50.4 equals almost 16 so the execution time of the grid appears to be scaling with the size of the grid which is as expected.

4.2 Results

The results for each terrain generation technique were grouped into several categories. Each technique was compared against each other in terms of the average framerate produced during the FRAPS benchmark. In addition, the mesh execution time of each technique was also recorded as well as a record of the CPU, GPU and memory usage throughout the test. Each test was carried out five times on a single tile of size 128x128, a single tile of size 512x512 and a 25x25 grid of tiles sized 128x128.

After all of the above, results of framerate and execution time were calculated to compare Fractal Noise using both Perlin Noise and Simplex Noise to compare the difference between the two techniques at a varying number of octaves. These results were recorded on a 5x5 grid of tiles sized 128x128. CPU, GPU and memory usage were also calculated for these tests.

Additionally, unless otherwise stated, each occurrence of Perlin Fractal Noise and Simplex Fractal Noise in the upcoming results graphs were generated using 10 octaves.

4.2.1 Average Framerate

For the average framerate test each run was setup with the terrain in front of the camera. The FRAPS benchmark was then started and throughout the duration of the entire benchmark the camera would slowly move forward across the terrain. The camera would move at the same speed and angle for each test to ensure fairness. Figure 26 below shows a visualisation of the testing method used.

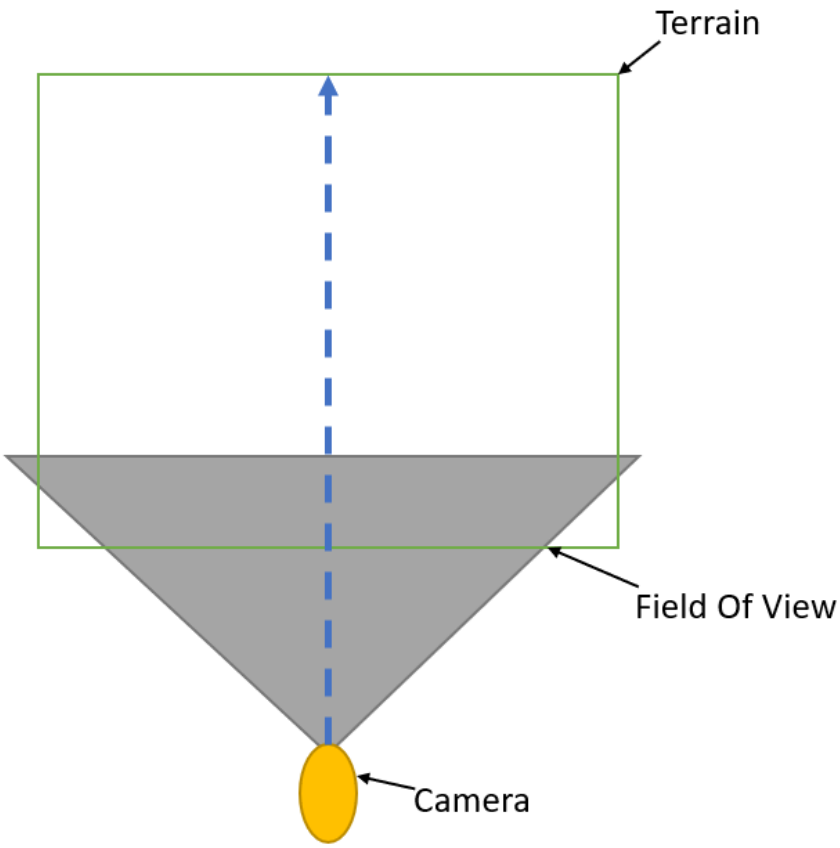


Figure 26. Visualisation of Testing Method

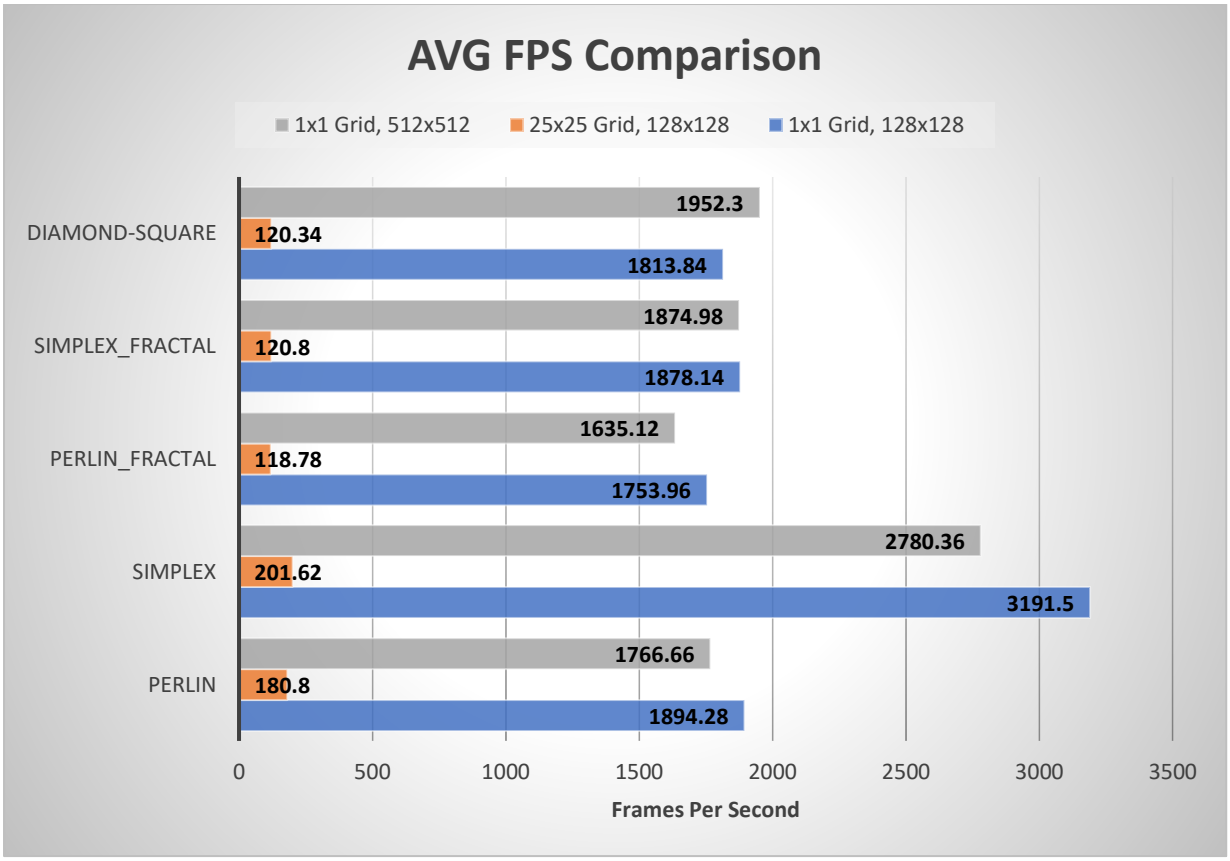


Figure 27. Framerate Comparison of Terrain Generation Techniques

The framerate test yielded some interesting results. The general pattern of the results is that the single tile sized 128x128 produced the highest framerate results followed by the single tile sized 512x512 and finally the 25x25 grid of 128x128 tiles. The performance difference between using a single tile compared to generating 25x25 tiles which is 625 tiles is huge with a decrease of about 90% across every technique. This shows the importance of tile management when implementing any of these terrain generation techniques for use in a video game as the framerate decreases extremely quickly as more and more tiles are generated without deleting any previous tiles.

First, when comparing each algorithm generating a single grid sized 128x128 the order of performance, in descending order of framerate, goes like this. Simplex, Simplex Fractal, Perlin, Diamond-Square and finally Perlin Fractal. Since the framerate for this test was in the thousands, anything below a change of about 100 frames per second can be regarded as in tolerance and therefore, about the same performance. This means there are only a few notable results in this test. Simplex Noise clearly outperforms the performance of every other technique in this test however, when using Fractal Noise with Simplex Noise the performance quickly drops down to be in line with the other techniques, but it still maintains an improvement over every technique other than Perlin Noise. This difference is only about 16 FPS though so realistically they are within the tolerance of about the same performance.

This trend, for the most part, continues for both a single tile of size 512x512 as well as 25x25 tiles of size 128x128. At 512x512 with a single tile Simplex Noise continues to outperform all

other techniques by a large amount, about a percentage increase of 55%. However, when using Simplex with Fractal Noise, only Diamond-Square outperforms it by about an increase of 4%.

At 25x25 tiles of 128x128 Simplex again outperforms all other techniques by an average of about 50% increase. This trend, however, does not continue when using Simplex with Fractal Noise as it performs worse than Perlin Noise and Simplex Noise which is to be expected since Simplex Fractal draws a more detailed looking terrain.

Evaluating all of these results shows that in terms of Frames Per Second, Simplex Noise is clearly the best performing in every test which means Simplex would be most suitable to use for a developer if they wish for an extremely fast method of generating noise without much added detail. If, however, a developer would like a more detailed noise function that performs slightly worse, then Simplex Fractal Noise is a great option.

Realistically, at this point, there is not enough information to decide which terrain generation technique is the outright best as even though Simplex performs extremely well, it may use much more system resources and take longer to execute than all of the other techniques, so further testing is required.

4.2.2 Execution Time

The execution time was recorded in exactly the same way for each technique. In the code a timer was placed at the beginning of the code loop for creating the mesh and was then stopped as soon as the full mesh was built. To ensure the test results were as reliable as possible this test was carried out a total of five times for each technique with the average of all five runs being used as the end result.

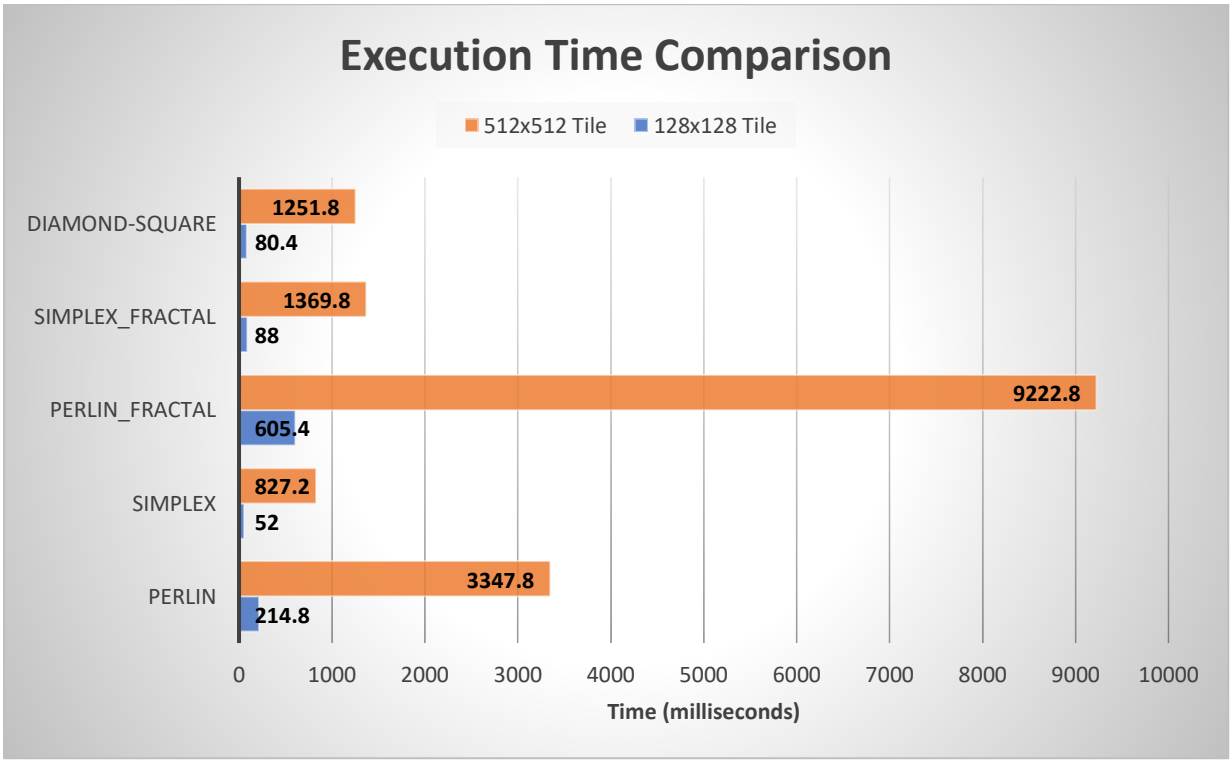


Figure 28. Time Comparison of Terrain Generation Techniques

As seen in Figure 28, there are some interesting results. First, Perlin Noise was extremely slow to execute. At first, this seemed like too much of an outlier therefore, the computer was restarted, and the test was carried out again, but the same results were acquired so this may be a serious problem with the original implementation of Improved Perlin Noise. This problem is even worse as the size of the tile increases and then furthered even more when using Perlin with Fractal Noise. This may be erroneous behaviour but as stated above, this test was carried out numerous times after a hardware restart. Because of this there is not much point including Perlin Noise in the reminder of the comparisons in this section as it is clearly outperformed by every other technique.

Moving on from Perlin, the fastest time to build the mesh out of all techniques at 128x128 was Simplex Noise at 52 milliseconds(ms). This was followed by Diamond-Square at 80ms and Simplex Fractal Noise at 88ms. At 512x512 this trend continues with exactly the same ordering. In terms of scaling from a 128x128 size tile to a 512x512 size tile each technique takes about 16 times more time to generate the larger tile which makes sense as the tile is 16 times larger than the smaller tile.

To evaluate these execution times, again Simplex is the best choice here in terms of speed. It runs quicker than all of the other techniques, so this is perfect for a developer looking for a noise function that executes as quickly as possible and is not worried about the detail of the output. For a more detailed output Simplex Fractal Noise is a very good choice here too as it executes only between 30ms and 40ms slower than Simplex Noise but provides a much more complex output in terms of the terrain generated.

4.2.3 CPU and GPU Usage

The CPU and GPU usage for each technique was calculated by taking the average usage from the beginning to end of each FRAPS framerate benchmark. As each benchmark was carried out a total of five times that also means these tests were carried out the same number of times to ensure the test results were as reliable as possible.

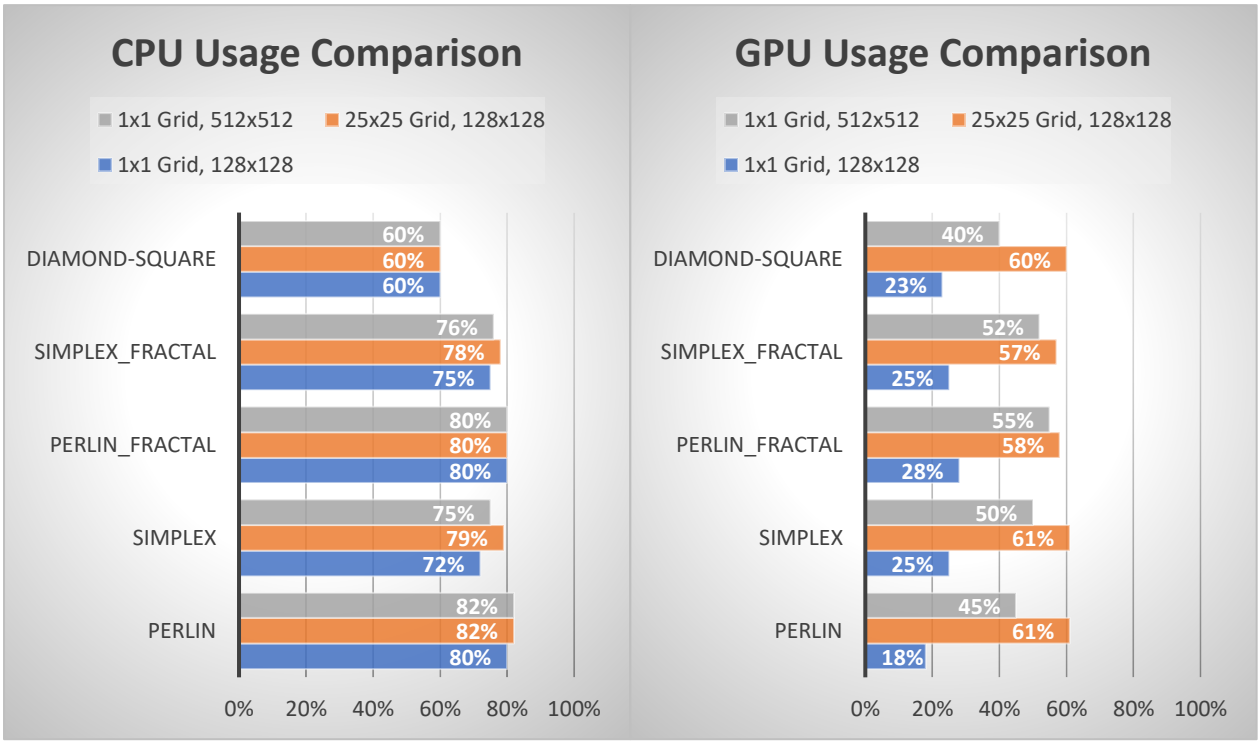


Figure 29. CPU and GPU Usage Comparison of Terrain Generation Techniques

The results in Figure 29 show that the CPU usage does not change all that much when scaling the number of tiles up or the size of tile up as all of the values recorded fall within only a few percent difference which can be seen as within tolerance. Potentially the only value of note here is that Simplex Noise used an extra 7% of CPU when running the 25x25 grid compared to a single tile. Additionally, most of the techniques used almost exactly the same amount of CPU usage, that being around 75%-80% with the exception of the Diamond-Square algorithm which only used 60% of the CPU across all three tests.

In terms of GPU usage each technique used, within tolerance, about the same amount of the GPU across each test. The only possible exception to this being Perlin Noise which used 18% of the GPU when drawing a single 128x128 tile which is 5% less usage than Diamond-Square and 10% less usage than Perlin with Fractal Noise. Other than this the amount of GPU used increased linearly as the amount of terrain to render increased. This leaves an expected and simple evaluation for GPU usage. In almost all cases, regardless of the technique used, the GPU usage will increase as the amount to render on screen also increases.

Evaluating CPU and GPU usage as a whole, most of the techniques utilise the CPU and GPU almost exactly the same with the exception of Diamond-Square which uses slightly less CPU than the other techniques. So, for the most cost effective in terms of CPU usage, Diamond-

Square should be the best choice however, in most cases the choice of any of these algorithms should not be determined by their CPU and GPU usage as they generally use about the same amount.

4.2.4 Memory Usage

Like the CPU and GPU usage tests, the memory usage test was carried out during the FRAPS benchmarks for calculating the framerate. The amount of memory used was taken directly from the Visual Studio diagnostic tool which displays, in Megabytes (MB), how much memory the currently running application is using.

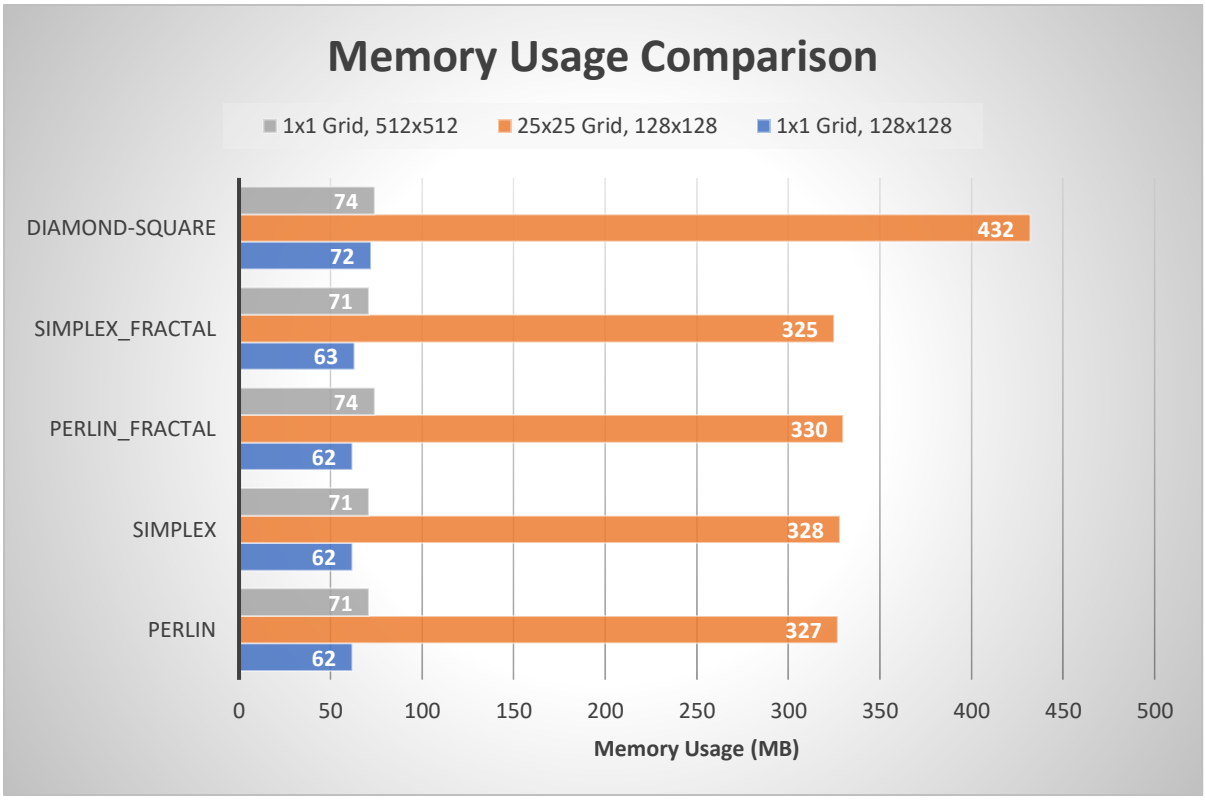


Figure 30. Memory Usage Comparison of Terrain Generation Techniques

The results of the memory usage tests show that as the size of terrain increases, so does the memory needed, which is expected. These results also show that the heightmap generation algorithm, which is Diamond-Square, generally uses more memory than the noise generation algorithms. This is expected because the Diamond-Square algorithm creates an entire 2D array of height values which is then copied into the mesh. This 2D array needs to be stored somewhere therefore, extra memory is used. The noise functions do not need to do this as they calculate each height value on the fly directly into the mesh Y value.

The interesting finding from these tests is that the difference in memory usage between the noise functions and the Diamond-Square algorithm begins to increase more and more as the number of tiles to generate also increases. The expected results for this test were that the amount of memory needed would increase, but it was never expected that it would increase as much as these results show.

To evaluate the memory usage of the techniques. If memory usage is important then Diamond-Square is not the terrain generation technique that should be used as it requires more memory than every other algorithm, especially when scaling up the amount to generate. For every other technique, the difference in memory usage is basically non-existent so if memory usage is the only issue, any noise function will perform extremely well.

4.2.5 Testing Octaves

After testing each terrain generation technique against each other, it was concluded that it may be beneficial to test the Fractal Noise techniques using a different number of octaves. This would give an insight into how the number of octaves affected performance.

These tests were carried out in the exactly same way as specified in the previous set of tests and results were taken for framerate, execution time, CPU usage, GPU usage and memory usage.

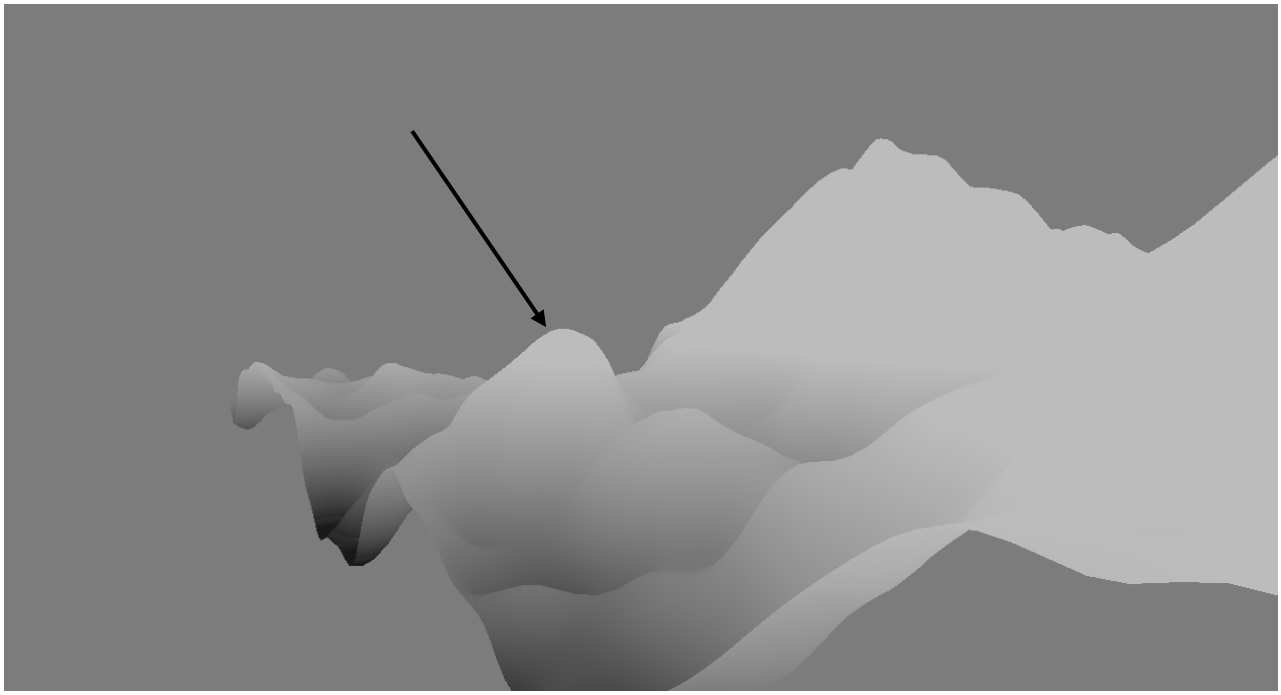


Figure 31. Perlin Fractal Noise at 5 Octaves

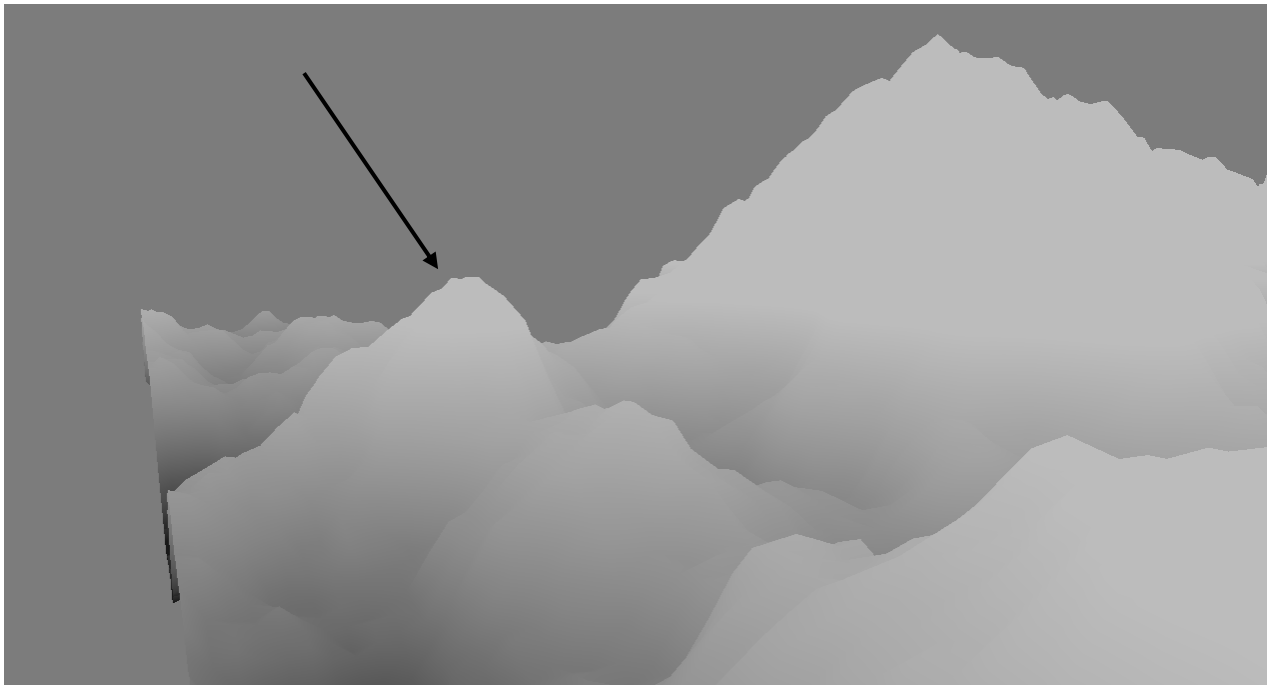


Figure 32. Perlin Fractal Noise at 10 Octaves

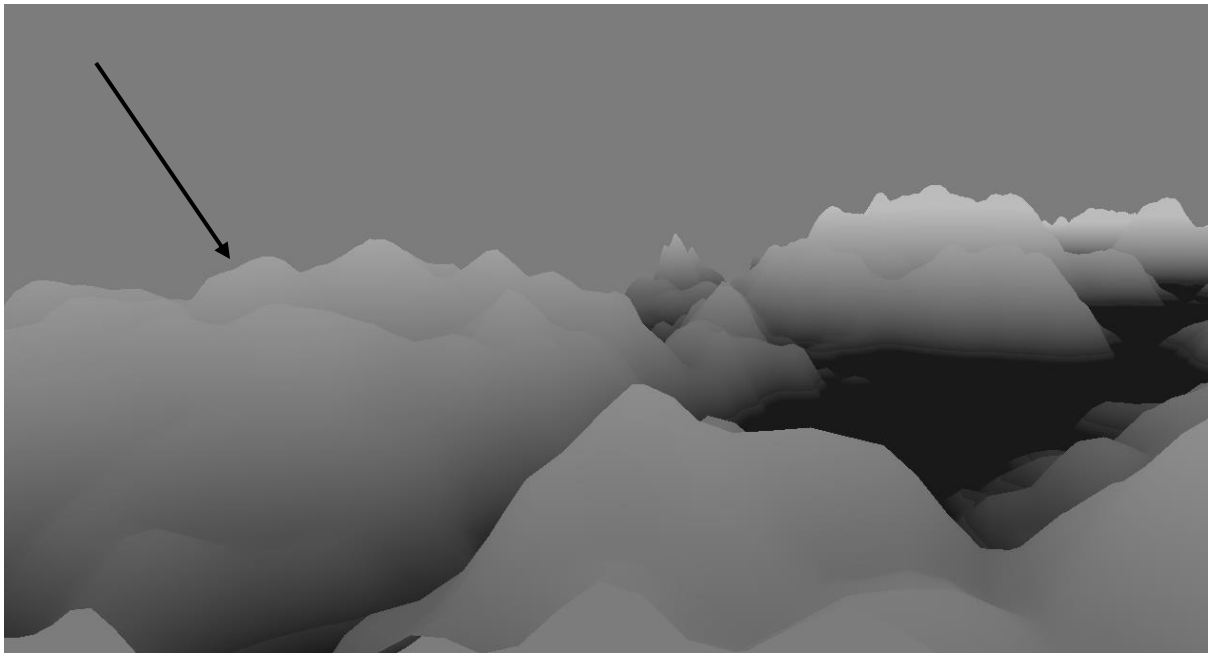


Figure 33. Simplex Fractal Noise at 5 Octaves

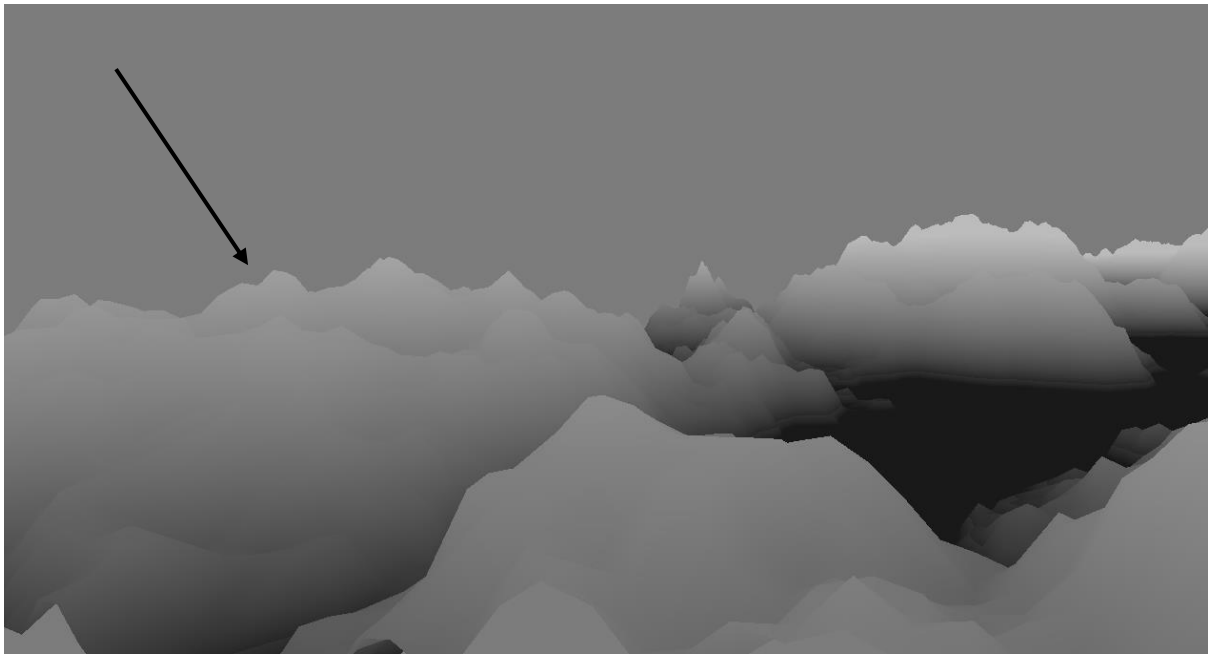


Figure 34. Simplex Fractal Noise at 10 Octaves

Figures 31 and 32 show the difference between Perlin Fractal Noise at 5 octaves and 10 octaves respectively while Figures 33 and 34 show the difference between Simplex Fractal Noise at 5 octaves and 10 octaves respectively. In general, the more octaves, the more detailed the terrain appears so Fractal Noise can be extremely useful therefore, it seems logical to test the differences in performance at these different octave levels.

4.2.5.1 Average Framerate

The average framerate for each of the tests was taken using the same method described in the framerate testing for the previous set of results. To recap, a benchmark was taken using FRAPS as the camera slowly flew across the centre of the generated terrain. This test was repeated five times.

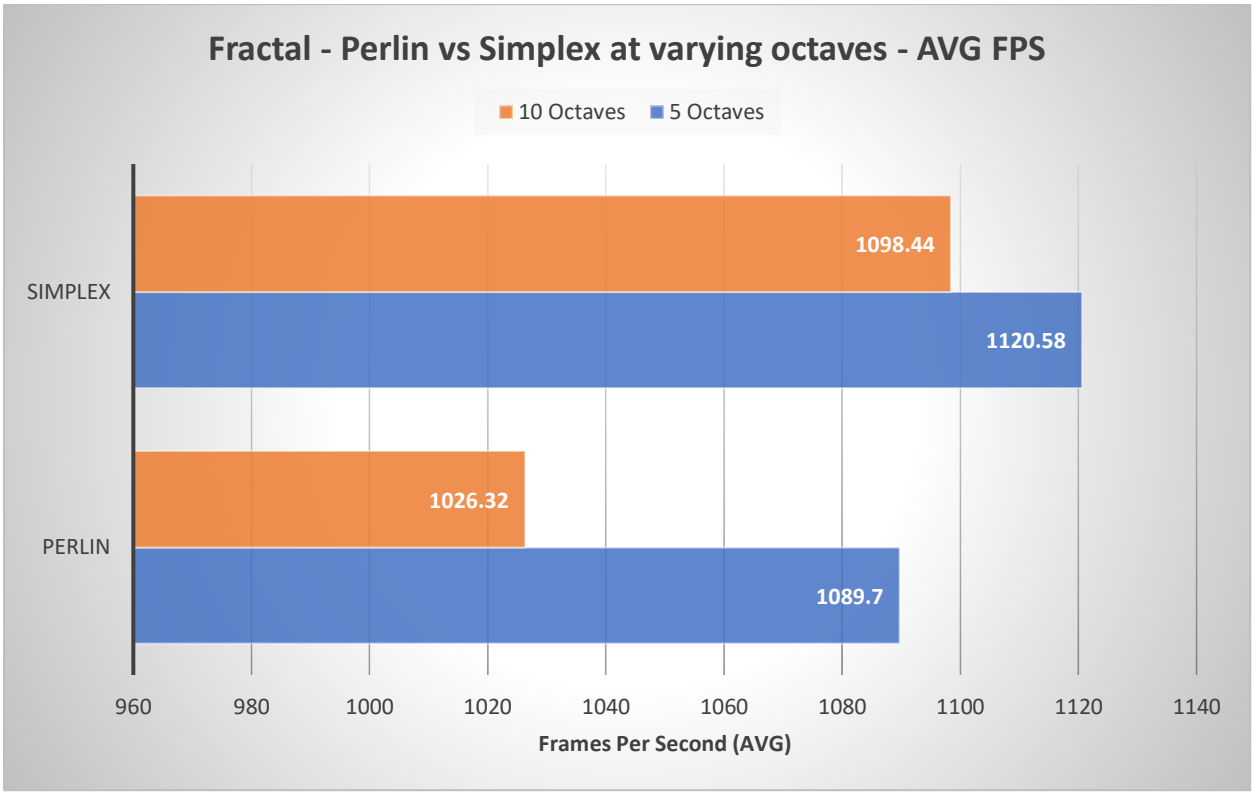


Figure 35. Fractal Noise Framerate Comparison Perlin vs Simplex

The results from Figure 35 show that Simplex Fractal Noise outperforms Perlin Fractal Noise by about 3% when using 5 octaves with a difference of about 7% when using 10 octaves. This shows that in general, Simplex Fractal performs better than Perlin Fractal especially as the scale of octaves increases. Meaning that if a developer is looking for a fractal noise function Simplex should be chosen over Perlin if framerate is the main requirement.

4.2.5.2 Execution Time

The execution time of each technique was measured in the same way stated above in the previous tests. A timer was placed in the code surrounding the loop used to generate the terrain mesh. This timer began at the start of the loop and ended as soon as the loop was completed therefore, giving the total time taken to generate the values for terrain tile.

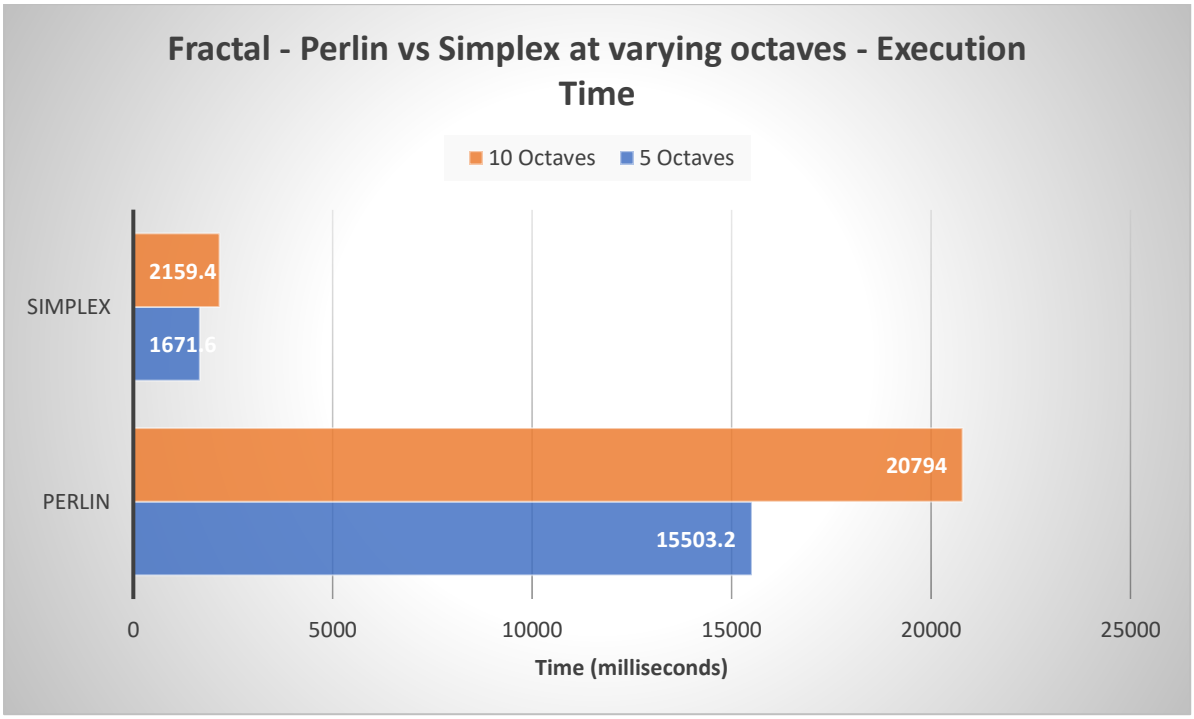


Figure 36. Fractal Noise Execution Time Comparison Perlin vs Simplex

Figure 36 shows an issue that appeared in the previous set of tests. Perlin noise takes an extremely long time to execute especially as the amount of detail, in this case the number of octaves increases. At 5 octaves Simplex Fractal is 828% faster than Perlin Fractal and at 10 octaves it is 863% faster. This test was carried out several times, even after a system restart to ensure the values were not being calculated incorrectly however, the results followed the same pattern every time.

This shows that if execution time is the most important factor, Simplex Fractal Noise is much quicker than that of Perlin Fractal Noise, especially as the level of detail increases.

4.2.5.3 CPU and GPU Usage

The CPU and GPU usage for this test was calculated using the same method outlined in the previous tests. The average usage was taken during the FRAPS framerate benchmark for both the CPU and GPU usage. This was repeated a total of five times.

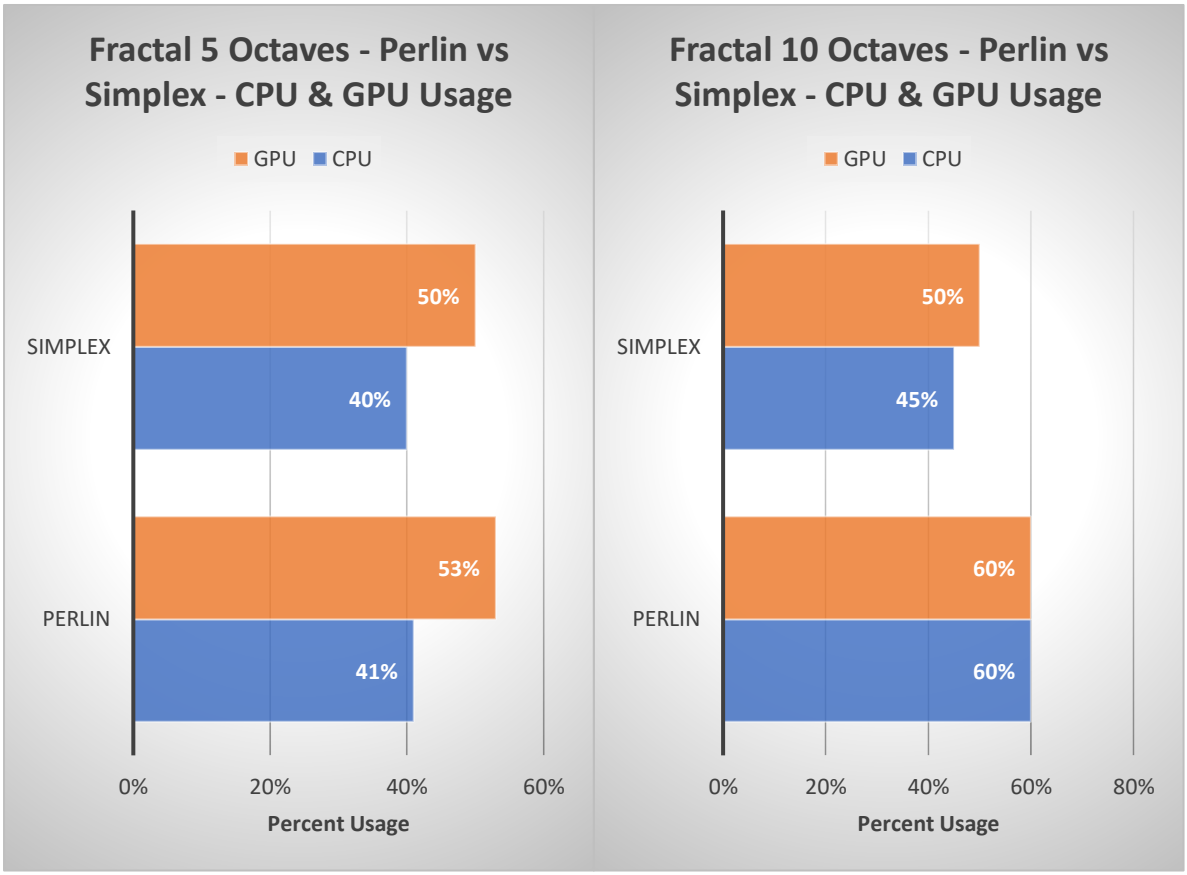


Figure 37. Fractal Noise Execution CPU & GPU Usage Perlin vs Simplex

The results from Figure 37 show that at 5 octaves, the CPU and GPU usage from both Perlin Fractal Noise and Simplex Fractal Noise are within 3% which is similar enough to regard them as about the same usage. This changes when moving to 10 octaves as Simplex Fractal does not increase the amount of CPU and GPU usage needed while the usage for Perlin Fractal Noise increases by 7% on the GPU and 19% on the CPU.

From these results it can be concluded that at a lower level of detail, Perlin Fractal and Simplex Fractal are both suitable if the main factor is CPU and GPU usage. However, as the number of octaves increases, Simplex Fractal Noise begins to pull ahead of Perlin Fractal by using less of both the CPU and GPU.

4.2.5.4 Memory Usage

Memory usage for this test was retrieved in the same way outlined in the previous tests. The diagnostic tool available in Visual Studio was used to get the value, in Megabytes, of the amount of memory being used by the currently running application.

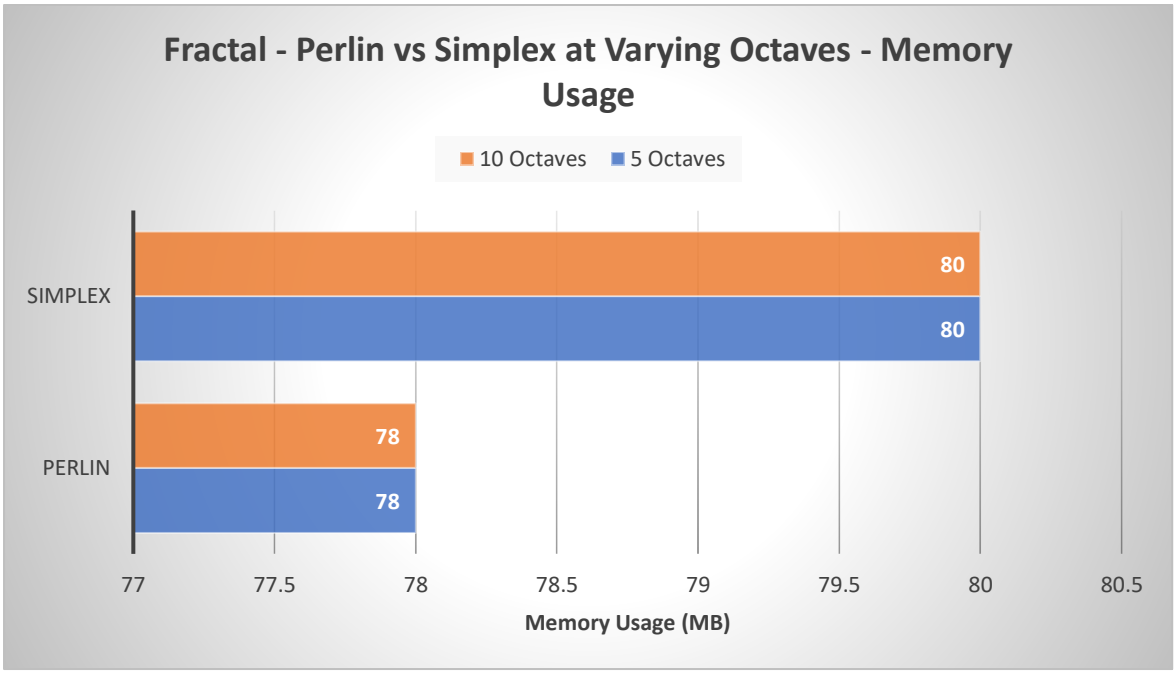


Figure 38. Fractal Noise Execution Memory Usage Perlin vs Simplex

Figure 38 shows that the memory requirements of Perlin Fractal Noise and Simplex Fractal Noise differ by about 2MB in favour of Perlin Fractal. This, however, is such low value that it can be considered that they, in general, use about the same amount of memory regardless of the number of octaves being calculated.

From these results it can be concluded that if the most important requirement for a fractal-based technique is memory usage, then either Perlin Fractal Noise or Simplex Fractal Noise will both perform perfectly well.

4.2.5.4 Overall

Using the results above it is fairly obvious that Simplex Fractal Noise should be chosen over Perlin Fractal Noise for several reasons. The most important of these reasons being that Simplex Fractal provides higher framerates than Perlin Fractal by about 5% average. In addition, the execution time of Simplex Fractal is a huge amount faster than Perlin Fractal. This execution time would be a problem in for example, a game where terrain was generated on the fly as the increased execution time from Perlin Noise could cause the game to stutter, especially if the memory of the application was not managed correctly.

4.2.6 Survey Results

The survey, which can be found in the appendix, was created using SurveyMonkey [34]. This survey was intended to give an idea of which terrain generation technique produces the most natural looking terrain as well as which looks most aesthetically pleasing. In this survey, questions 1 and 3 were the same question and questions 2 and 4 were the same question with the only difference being questions 1 and 2 used a different set of images to questions 3 and 4. This was to see if the results would remain the same even if the techniques generated a different looking tile of terrain.

Survey questions:

1 & 3: In your opinion, which of the following images most resembles mountainous terrain?

2 & 4: In your opinion, which of the following images is most aesthetically pleasing?

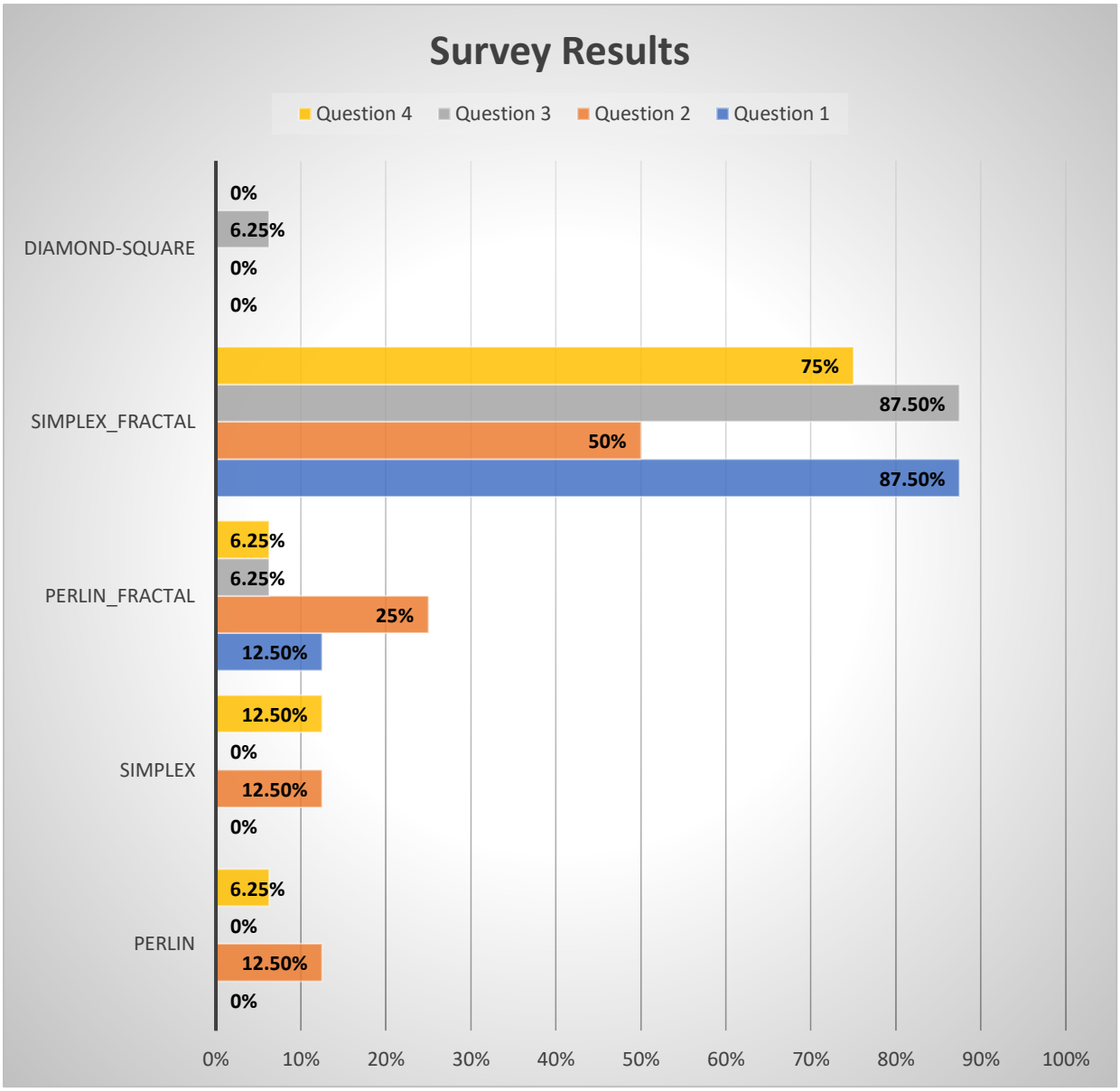


Figure 39. Survey Results

Figure 43 shows the results obtained from a survey with 32 participants. In this case, it makes most sense to compare the results of question 1 with question 3 as they were the same question and separately compare the results of questions 2 and 4 as they were the same question.

Questions 1 and 3 asked the participant which image they found to resemble mountainous terrain the most and the results were very similar. In both cases, Simplex Fractal Noise received the most votes with 87.5% with Perlin Fractal receiving 12.5% in question 1 and 6.25% in question 3 with the other 6.25% going to Diamond-Square. These results make a lot of sense since the Fractal Noise techniques provide a lot more detail to the terrain so it was obvious that they would come out on top. It also makes sense that Simplex Fractal beat Perlin Fractal since in both images, Simplex Fractal offers a much greater variation of high and low terrain while Perlin Fractal has quite a lot of high terrain.

Questions 2 and 4 asked the participant which image they found the most aesthetically pleasing, this was a deliberately broad question simply to see how much the results would vary between each participant. The results show that again Simplex Fractal produced the most aesthetically pleasing image overall, receiving 50% and 75% of votes in questions 2 and 4 respectively. The remaining votes were spread across Perlin Fractal, Perlin and Simplex Noise. This was extremely interesting because there was no expectation for Perlin Noise or Simplex Noise to receive any votes for any of the questions since they lack much of the detail from the other techniques however, 6.25% of people found Perlin most aesthetically pleasing in question 4 while 12.5% of people found this to be Simplex which is very surprising.

From all of these results it can be concluded that the technique which is both most aesthetically pleasing and has the most natural potential for creating appealing mountainous terrain is the Simplex Fractal Noise technique by quite a large margin.

4.2.7 Evaluating the Results

Looking at all of the results retrieved during the entirety of the testing phase it is fairly obvious that Simplex Fractal Noise is overall, the most suitable terrain generation method in many cases. This is because it performs only slightly worse than techniques such as Simplex and Diamond-Square in terms of average framerate and execution time however, Simplex Fractal can provide much more detailed terrain for that extra cost in performance. CPU and GPU usage remain almost equal between techniques and the same goes for memory usage with the exception of the Diamond-Square algorithm which uses much more memory.

To conclude, if a developer is looking for a flexible noise algorithm then Simplex Fractal Noise may well be their best bet other than in very specific circumstances. For example, if a developer is simply looking for the best performing technique with no need for extra detail, then standard Simplex Noise would be the best choice in this case. Every other technique tested throughout this project simply could not perform as well as Simple Noise and could not produce the detail of Simplex Fractal Noise at the same level of performance and resource usage.

Chapter 5. Conclusion

To conclude this dissertation, it seems necessary to reflect on the project as a whole and look back on what was learnt throughout the process of creating this project, ending on how the project could potentially be improved if it were to be carried out again or further developed in the future.

5.1 Project Aim

At the very beginning of this dissertation the aim for the project was outlined along with a few select objectives that would need to first be met before the aim could be achieved. To recap, the aim was "To compare and evaluate commonly used techniques for real-time terrain generation."

To achieve the project aim, the following objectives would first need to be met:

- Investigate commonly used real-time terrain generation techniques.
- Implement an efficient method of creating meshes suitable for terrain generation.
- Apply terrain generation techniques to a suitable mesh.
- Collect, compare and evaluate performance and effectiveness of the discovered techniques.

The first objective was met during the background research section of this dissertation in which research was carried out into the various terrain generation techniques ultimately used for this project. All of these techniques were commonly used at some point in time within the video games industry.

The second objective was met and explained during the design and implementation section of this dissertation. In this section a description is included of how a base mesh was created for use with a variety of terrain generation techniques and how that mesh code would be used to apply the techniques used for this project.

The third objective was met during the implementation of the project code. The implementation of each of these techniques is explained during the implementation section of this dissertation along with an accompanying image of the output when running the code.

The fourth and final objective was met throughout different stages of the project. Collecting the results data for each of the techniques was met during the testing phase of the project while the comparison and evaluation of these results were achieved during the testing and results evaluation section of this dissertation.

To achieve the aim of the project, first each objective would need to be achieved. The explanation for how each of these objectives were achieved above provide enough evidence to confidently say that this project has achieved the original aim "To compare and evaluate commonly used techniques for real-time terrain generation."

5.2 What was Learnt

It is always important to reflect on what was learnt throughout a project, especially one as large as this to help evaluate if the project kept the focus of the original aim and it also helps ensure anything learnt can be carried on to any future projects.

5.2.1 Results from Testing

During the conclusion of the results section it was decided that the best technique overall was Simplex Fractal Noise which combines Simplex Noise at increasing frequencies and decreasing amplitudes to create the most detailed, natural looking terrain out of all of the tested techniques while achieving a good level of performance. To give a better idea of which technique to use depending entirely on the following specific requirements.

Average Framerate: Simplex Noise.

Execution Time: Simplex Noise.

CPU Usage: Diamond-Square Algorithm.

GPU Usage: Every technique performed the same within tolerance.

Memory Usage: Every technique performed the same within tolerance except Diamond-Square which utilises more memory.

Visual Output: Simplex Fractal Noise.

Overall based on all previous categories: Simplex Fractal Noise.

5.2.2 Teachings from the Project

Throughout the project much was learnt. Research into the terrain generation algorithms resulted in invaluable knowledge into how video game developers create the illusion of an immersive game world and how terrain generation can directly enhance that process. In addition, it was extremely interesting to research how each of the used terrain generation techniques can be implemented, used and even tweaked further based on the specific needs of the developer.

There were several programming practices learnt from the carrying out this project which included, always ensure to decide on a basic structure for code before beginning, especially if the project is intended have a fair amount of code. It has been learnt that migrating such a large amount of code into classes after the implementation is almost complete is not a good idea as it can result in a whole list of problems as well as a need to change a good portion of the implementation.

Perhaps most importantly, this project has met the specific aim set out at the beginning of the project and has been able to find the most suitable terrain generation techniques for many specific situations.

5.3 Future Improvements

The project could absolutely be improved in the future. This could be by improving the project code to allow tiles to seamlessly generate infinitely in the direction that the camera is moving. This problem was almost solved and implemented into the project, but time restraints prevented it from being completed.

Another improvement for the future could be to implement a few different implementations of Perlin Noise as the current implementation in this project takes an extremely long time to execute in comparison to the other techniques. During the testing phase the tests for this were carried out several times after a computer restart and the same results were achieved so there may be an issue with the implementation of the Perlin Noise function.

Of course, it would be possible to improve this project by testing even more terrain generation techniques after further research. This would broaden the current data set and provide opportunities for even more comparison between techniques. The testing phase could also be improved by testing each technique at more varying levels of tile size as this project focused exclusively on tile sizes of 128x128 and 512x512. Testing at even more tile sizes would introduce further comparison in terms of how each technique scales with tile size.

References

[1] Togelius J, Yannakakis G, Stanley K, Browne C. Search-Based Procedural Content Generation: A Taxonomy and Survey. IEEE Transactions on Computational Intelligence and AI in Games [Internet]. [cited 11 April 2019];172-186. Available from: https://ieeexplore.ieee.org/abstract/document/5756645

[2] Mojang. [Internet]. [cited 11 April 2019]. Available from: https://www.minecraft.net/en-us/article/minecraft-snapshot-18w32a

[3] Gustavson S. [Internet]. [cited 16 April 2019]. Available from: http://webstaff.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf

[4] [Internet]. Web.archive.org. [cited 16 April 2019]. Available from: https://web.archive.org/web/20071008165504/http://www.noisemachine.com/talk1/6.html

[5] [Internet]. [cited 16 April 2019]. Available from: https://web.archive.org/web/20010122043500/http://www.noisemachine.com/talk1/6.html

[6] Perlin K. [Internet]. [cited 16 April 2019]. Available from: https://mrl.nyu.edu/~perlin/doc/vase.html

[7] Simplex Noise - Procedural Content Generation Wiki [Internet]. Pcg.wikidot.com. [cited 16 April 2019]. Available from: http://pcg.wikidot.com/pcg-algorithm:simplex-noise

[8] [Internet]. [cited 16 April 2019]. Available from: https://clojurefun.files.wordpress.com/2012/08/2012-08-04-0-3-simplex-noise.png

[9] Rombauts S. SRombauts/SimplexNoise [Internet]. GitHub. [cited 16 April 2019]. Available from: https://github.com/SRombauts/SimplexNoise

[10] SIGER STUDIO. [Internet]. [cited 16 April 2019]. Available from: https://www.sigershop.eu/wp-content/uploads/2016/07/sn_octave.png

[11] www.3d-map-generator.com | 3D Map Generator – Terrain – How it works [Internet]. 3d-map-generator.com. [cited 17 April 2019]. Available from: https://www.3d-map-generator.com/3d-map-generator-terrain-how-it-works/

[12] [Internet]. [cited 17 April 2019]. Available from: https://upload.wikimedia.org/wikipedia/commons/5/57/Heightmap.png

[13] Midpoint Displacement Algorithm - Procedural Content Generation Wiki [Internet]. Pcg.wikidot.com. [cited 17 April 2019]. Available from: http://pcg.wikidot.com/pcg-algorithm:midpoint-displacement-algorithm

[14] Diamond-Square Algorithm - Procedural Content Generation Wiki [Internet]. Pcg.wikidot.com. [cited 17 April 2019]. Available from: http://pcg.wikidot.com/pcg-algorithm:diamond-square-algorithm

[15] [Internet]. [cited 17 April 2019]. Available from: https://upload.wikimedia.org/wikipedia/commons/b/bf/Diamond_Square.svg

[16] Bjarne Stroustrup's Homepage [Internet]. Stroustrup.com. [cited 15 April 2019]. Available from: http://www.stroustrup.com/

[17] History of C++ - C++ Information [Internet]. Cplusplus.com. [cited 12 April 2019]. Available from: http://www.cplusplus.com/info/history/

[18] A Brief Description - C++ Information [Internet]. Cplusplus.com. [cited 12 April 2019]. Available from: http://www.cplusplus.com/info/description/

[19] OpenGL Overview [Internet]. Opengl.org. [cited 12 April 2019]. Available from: https://www.opengl.org/about/

[20] The Khronos Group [Internet]. The Khronos Group. [cited 12 April 2019]. Available from: https://www.khronos.org/

[21] Computing, School of - Computing, School of - Newcastle University [Internet]. Ncl.ac.uk. [cited 15 April 2019]. Available from: https://www.ncl.ac.uk/computing/

[22] Visual Studio IDE, Code Editor, Azure DevOps, & App Center - Visual Studio [Internet]. Visual Studio. [cited 15 April 2019]. Available from: https://visualstudio.microsoft.com/

[23] FRAPS game capture video recorder fps viewer [Internet]. Fraps.com. [cited 15 April 2019]. Available from: http://www.fraps.com/

[24] TechPowerUp [Internet]. TechPowerUp. [cited 15 April 2019]. Available from: https://www.techpowerup.com/gpuz/

[25] HWMONITOR | Softwares | CPUID [Internet]. Cpuid.com. [cited 15 April 2019]. Available from: https://www.cpuid.com/softwares/hwmonitor.html

[26] [Internet]. [cited 24 April 2019]. Available from: https://cdn-images-1.medium.com/max/1600/1*Fzz56Ps_vQSTPHDO9IomXw.png

[27] Face Culling - OpenGL Wiki [Internet]. Khronos.org. 2019 [cited 17 April 2019]. Available from: https://www.khronos.org/opengl/wiki/Face_Culling#Winding_order

[28] sol-prog/Perlin_Noise [Internet]. GitHub. [cited 18 April 2019]. Available from: https://github.com/sol-prog/Perlin_Noise

[29] Perlin K. Improving noise. Proceedings of the 29th annual conference on Computer graphics and interactive techniques - SIGGRAPH '02 [Internet]. 2002 [cited 18 April 2019]. Available from: https://mrl.nyu.edu/~perlin/paper445.pdf

[30] Learning how Perlin noise works [Internet]. Huttar.net. [cited 18 April 2019]. Available from: http://www.huttar.net/lars-kathy/graphics/perlin-noise/perlin-noise.html

[31] Perlin K. Noise Hardware [Internet]. [cited 18 April 2019]. Available from: https://www.csee.umbc.edu/~olano/s2002c36/ch02.pdf

[32] Perlin noise / fuzzy notepad [Internet]. Eev.ee. [cited 18 April 2019]. Available from: https://eev.ee/blog/2016/05/29/perlin-noise/

[33] VadimDev/Unreal-Engine-diamond-square-algorithm [Internet]. GitHub. [cited 19 April 2019]. Available from: https://github.com/VadimDev/Unreal-Engine-diamond-square-algorithm

[34] SurveyMonkey: The World's Most Popular Free Online Survey Tool [Internet]. Surveymonkey.com. [cited 29 April 2019]. Available from: https://www.surveymonkey.com

Glossary

API – Application programming Interface. A collection of functions which aid in the creation of applications.

FPS – Frames Per Second. The number of images output by the graphics card every second.

IDE – Integrated Development Environment. A software environment that provides necessary tools for software developers to develop software.

Local Space – Vertex coordinates for a model are defined in relation to a local origin.

Matrix – A set of numbers commonly used in computer graphics to transform a model from local space to screen space.

OOP – Object Oriented Programming. A method of computer programming based on the concept of data and functions being stored and accessed through objects.

OpenGL – Open Graphics Library. The graphics API specification used to enable graphics to be drawn to screen.

Pseudo-random – Predictable, controlled randomness whereby the same input will produce the same output.

Screen Space – Vertex positions are defined by their exact on-screen location.

Texture Coordinates – An attribute usually associated with a vertex for use when applying textures to a model.

Vertex – A point where two or more lines meet. Specifically, in computer graphics a vertex is a data structure which specifies the position of a point in space.

Appendix

All Testing Results

Perlin Framerate

Perlin Noise Performance	128x128					
Tiles	1			25x25		
Test Number	Framerate(min)	Framerate(max)	Framerate(avg)	Framerate(min)	Framerate(max)	Framerate(avg)
1	1859	1972	1922.4	178	180	178.8
2	1835	1936	1885.7	177	181	178.9
3	1775	1934	1861.2	178	181	180
4	1805	1978	1934.6	179	182	180.5
5	1758	1897	1867.5	185	187	185.8
Average ->	1806.4	1943.4	1894.28	179.4	182.2	180.8

Perlin Noise Performance	512x512		
Tiles	1		
Test Number	Framerate(min)	Framerate(max)	Framerate(avg)
1	1384	1909	1626.7
2	1485	1934	1774.8
3	1499	1952	1702.2
4	1102	1937	1978.3
5	1499	1830	1751.3
Average ->	1393.8	1912.4	1766.66

Perlin Execution Time

Perlin Noise Execution	**128x128**	**512x512**
Test Number	Value(milliseconds)	Value(milliseconds)
1	217	3367
2	218	3331
3	210	3348
4	209	3345
5	220	3348
Average Time ->	**214.8**	**3347.8**

Simplex Framerate

Simplex Noise Performance	128x128					
Tiles	1			25x25		
Test Number	Framerate(min)	Framerate(max)	Framerate(avg)	Framerate(min)	Framerate(max)	Framerate(avg)
1	2998	3349	3274.6	204	207	205.7
2	2988	3363	3299.1	197	199	198.1
3	3004	3363	3301	198	201	199.9
4	2964	3368	3279.3	202	205	203.8
5	2406	3347	2803.5	200	202	200.6
Average ->	2872	3358	3191.5	200.2	202.8	201.62

Simplex Noise Performance	512x512		
Tiles	1		
Test Number	Framerate(min)	Framerate(max)	Framerate(avg)
1	2927	3027	2986
2	2808	3036	2975.7
3	2917	3051	2987
4	2651	3037	2911.1
5	1949	2074	2042
Average ->	2650.4	2845	2780.36

Simplex Execution Time

Simplex Noise Execution	**128x128**	**512x512**
Test Number	Value(milliseconds)	Value(milliseconds)
1	53	838
2	52	824
3	52	826
4	52	823
5	51	825
Average Time ->	**52**	**827.2**

Perlin Fractal Framerate

Fractal Perlin Noise Performance	128x128					
Tiles	1			25x25		
Test Number	Framerate(min)	Framerate(max)	Framerate(avg)	Framerate(min)	Framerate(max)	Framerate(avg)
1	1002	2355	1684.7	117	121	118.8
2	1960	2385	1917.8	117	121	118.8
3	1032	2401	1825.4	116	121	118.5
4	1444	1677	1590.2	116	121	118.8
5	1376	1326	1751.7	117	122	119
Average ->	1362.8	2028.8	1753.96	116.6	121.2	118.78

Fractal Perlin Noise Performance	512x512		
Tiles	1		
Test Number	Framerate(min)	Framerate(max)	Framerate(avg)
1	1075	1964	1561.2
2	1891	2041	1552.5
3	1499	1734	1789.7
4	1542	1752	1584.8
5	1479	1742	1687.4
Average ->	1497.2	1846.6	1635.12

Perlin Fractal Execution Time

Fractal Perlin Noise Execution	**128x128**	**512x512**
Test Number	Value(milliseconds)	Value(milliseconds)
1	606	9207
2	606	9229
3	607	9222
4	608	9221
5	600	9235
Average Time ->	**605.4**	**9222.8**

Simplex Fractal Framerate

Fractal Simplex Noise Performance	128x128					
Tiles	1			25x25		
Test Number	Framerate(min)	Framerate(max)	Framerate(avg)	Framerate(min)	Framerate(max)	Framerate(avg)
1	1806	1923	1892.4	117	121	119
2	1678	1907	1866.2	121	123	122
3	1811	1911	1887.9	118	123	119.9
4	1823	1923	1899.1	120	123	121.4
5	1761	1890	1845.1	120	123	121.7
Average ->	1775.8	1910.8	1878.14	119.2	122.6	120.8

Fractal Simplex Noise Performance	512x512		
Tiles	1		
Test Number	Framerate(min)	Framerate(max)	Framerate(avg)
1	1705	1854	1815.7
2	1683	1978	1863.7
3	1869	1986	1955.1
4	1767	1902	1855.4
5	1806	1932	1885
Average ->	1766	1930.4	1874.98

Simplex Fractal Execution Time

Fractal Simplex Noise Execution	**128x128**	**512x512**
Test Number	Value(milliseconds)	Value(milliseconds)
1	89	1398
2	87	1361
3	88	1364
4	88	1363
5	88	1363
Average Time ->	**88**	**1369.8**

Diamond-Square Framerate

Diamond Square Performance	128x128					
Tiles	1			25x25		
Test Number	Framerate(min)	Framerate(max)	Framerate(avg)	Framerate(min)	Framerate(max)	Framerate(avg)
1	1634	1952	1868	118	121	119.8
2	1653	1777	1751	118	123	120.1
3	1653	1774	1742	116	121	119.1
4	1897	1948	1929.9	119	123	121.4
5	1704	1808	1778.3	120	123	121.3
Average ->	1708.2	1851.8	1813.84	118.2	122.2	120.34

Diamond Square Performance	512x512		
Tiles	1		
Test Number	Framerate(min)	Framerate(max)	Framerate(avg)
1	1924	1984	1961.8
2	1877	1976	1952.5
3	1838	1975	1942
4	1879	1975	1950.9
5	1886	1974	1954.3
Average ->	1880.8	1976.8	1952.3

Diamond-Square Execution Time

Diamond Square Execution	**128x128**	**512x512**
Test Number	Value(milliseconds)	Value(milliseconds)
1	80	1248
2	80	1251
3	82	1251
4	80	1255
5	80	1254
Average Time ->	**80.4**	**1251.8**

CPU Usage

CPU Usage	128		512
	1x1	25x25	1x1
Perlin	80%	82%	82%
Simplex	72%	79%	75%
P_Fractal	80%	80%	80%
S_Fractal	75%	78%	76%
D_Square	60%	60%	60%

GPU Usage

GPU Usage	128		512
	1x1	25x25	1x1
Perlin	18%	61%	45%
Simplex	25%	61%	50%
P_Fractal	28%	58%	55%
S_Fractal	25%	57%	52%
D_Square	23%	60%	40%

Memory Usage

RAM Usage (MB)	128		512
	1x1	25x25	1x1
Perlin	62	327	71
Simplex	62	328	71
P_Fractal	62	330	74
S_Fractal	63	325	71
D_Square	72	432	74

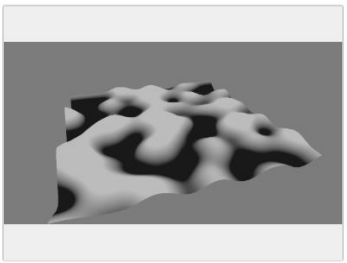
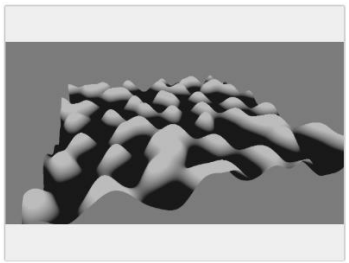
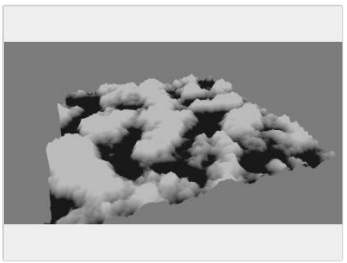
Fractal Noise Octave Testing

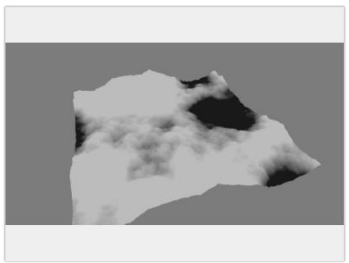
Fractal Octave Performance	Perlin Fractal 5x5				Simplex Fractal 5x5			
Octaves	5		10		5		10	
Test Number	Framerate(avg)	Time(milliseconds)	Framerate(avg)	Time(milliseconds)	Framerate(avg)	Time(milliseconds)	Framerate(avg)	Time(milliseconds)
1	1172.4	15635	1049.2	20802	1183.8	1657	1122.2	2160
2	1086.1	15395	1045.7	20647	1009.3	1633	1018.1	2183
3	1173.5	15485	1100	20971	1187.2	1699	1102.2	2122
4	1009.9	15569	978.4	21046	1041.6	1713	1081.6	2146
5	1006.6	15432	958.3	20504	1181	1656	1168.1	2186
Average ->	1089.7	15503.2	1026.32	20794	1120.58	1671.6	1098.44	2159.4
CPU ->	41%		60%		40%		45%	
GPU ->	53%		60%		50%		50%	
RAM(MB) ->	78		78		80		80	

Survey

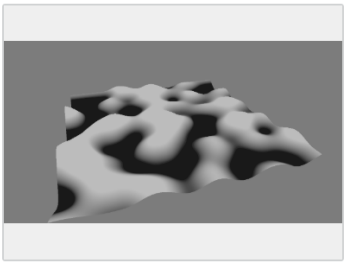
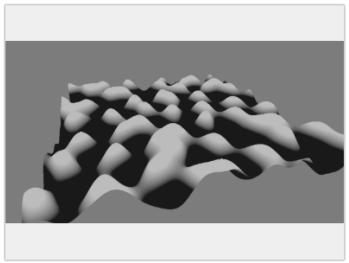
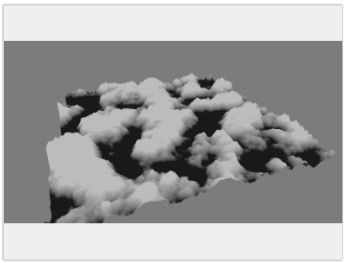
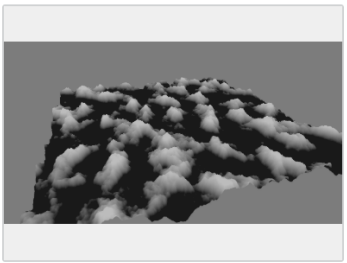
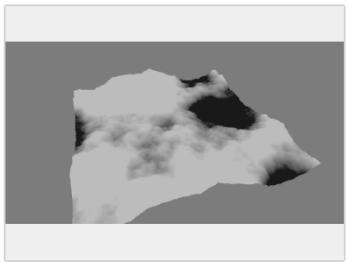
Terrain Generation Dissertation Survey

* 1. In your opinion, which of the following images most resembles a mountainous terrain?

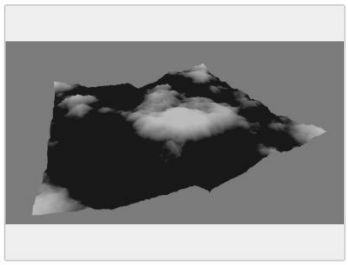
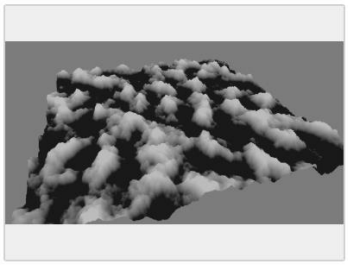
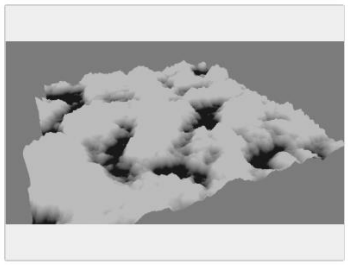
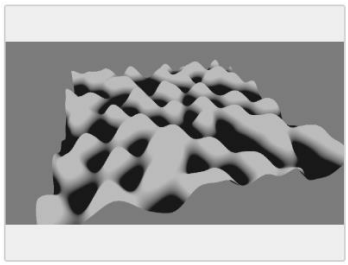
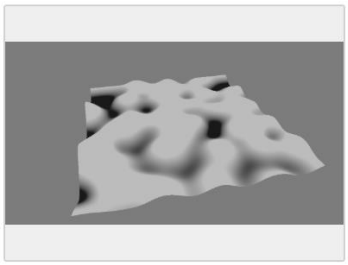






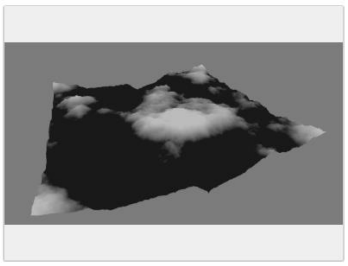
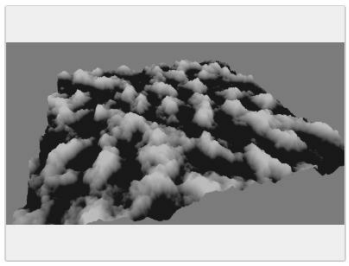
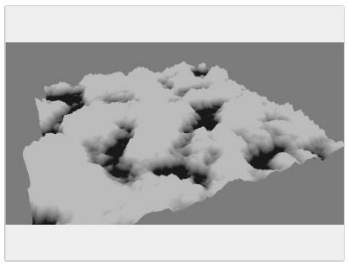
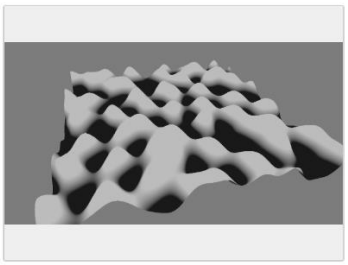
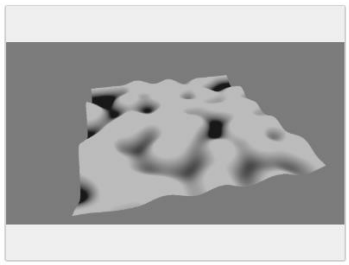
2. In your opinion, which of the following images is most aesthetically pleasing?

3. In your opinion, which of the following images most resembles mountainous terrain?

4. In your opinion, which of the following images is most aesthetically pleasing?

Done