Real-Time Fractal Landscapes Fly Over
Final Report

Submitted for the BSc in
Computer Science with Games Programming

May 2016

by

Sam Murphy

Word Count: 9975

Table of Contents

1 Introduction

Visualizing terrain is an important part of many fields, such as the film and gaming industries. One method of generating terrain is fractal landscapes; it has been used since the eighties in films and has continued to rise in popularity in the games industry as well. Often in films and some games the terrain generation is pre-computed rather than real-time, and while this tends to produce terrain of higher quality, performance and storage space are sacrificed. The alternative to this is to generate terrain in real-time or procedurally as the user moves around the world.

Terrain is defined as a stretch of land, in particular the surface features of that stretch. This includes a number of things; such as how mountainous or smooth the terrain is, the kind of foliage that grows in the areas, as well ground the ground covering be it grass, rock, or sand. Even weather can affect how we view a terrain, obscuring it fog or covering it with snow.

As the generation of terrain covers a vast number of components, this project will mostly focus on the generation of the topology itself with a simple simulation allowing the user to fly over the landscape. This report summarizes fractal landscapes, the technologies and algorithms associated with them, and the design and development of the project itself.

2 Aim and Objectives

The aim of this project is to design and implement a piece of software that will generate a complex fractal landscape in real-time and incorporate a simple simulation of a plane flying over the landscape.

This aim will be considered achieved when the following objectives have been completed:

1. Generate 3D terrain in real-time using fractal algorithms.
2. Create a simple user controlled simulation of a flying plane to navigate the landscape.
3. Enable the user to have basic control over the generation of the landscape.
4. Create an easy to use user interface to provide the user with performance analysis, and control over the landscape.
5. The generated terrain should include additional details, such as trees and lakes.

Objective 1 – Generate 3D Terrain in Real-Time

Sub Objective 1.1 Realism

The generated terrain should be convincing enough that it could exist somewhere in the world. This should be achieved by comparing the output of multiple different fractal algorithms.

Sub Objective 1.2 Real-Time

In order to complete this objective, the terrain should run in real-time, which shall be considered as 30 frames per second of higher.

Sub Objective 1.3 Textured

The terrain must also be textured appropriately through the use of shaders, preferably using generated textures so no images have to be stored.

Sub Objective 1.4 Infinite

The terrain must generate infinitely as the player moves around, or at the very least large enough that the player would struggle to find the edge.

Objective 2 – Simple Simulation of Flying Plane

Sub Objective 2.1 Plane Model

The user should start in a model of a plane, loaded from a model file and render using standard methods.

Sub Objective 2.2 User Controls

The plane should be controlled through simple, intuitive controls to allow the user to alter the direction and altitude of the plane.

Sub Objective 2.3 Camera

The camera should be locked in position in a third person perspective of the plane, making sure that enough of the terrain is visible around the plane. Other camera options should also be available for debugging and testing purposes.

Objective 3 – User Control of Terrain

In order to complete this objective, the application should enable the user to have some control over the generation of the terrain, this should include but is not limited to the ability to select how mountainous or flat the terrain is, a way of completely regenerating the terrain without restarting the application, and a method of saving and loading previously generated terrain.

Objective 4 – Graphic User Interface

Sub Objective 4.1 Terrain Controls

A successful GUI should allow the user to be able to easily make the previously discussed changes to the terrain generation process through a simple and easy to understand UI.

Sub Objective 4.2 Performance Statistics

The GUI should show the user the relevant details for performance analysis, such as the average frames per second, the number of sections of terrain that have been generated and the number that are being rendered.

Objective 5 – Additional Terrain Details

Sub Objective 5.1 Vegetation

The terrain should feature trees and possibly other plant life, grouped together in forests and individually in realistic looking positions. This means that trees should only appear where the gradient of the ground is suitably level.

Sub Objective 5.2 Water

The terrain should feature water in varying sizes, taking the form of lakes and if possible oceans. These could be achieved with a simple sea level variable, and if most other objectives have been achieved could be extended to include realistic flowing rivers and streams although this would likely be more complex to implement.

Sub Objective 5.3 Atmospheric Details

The world should include a skybox to help immerse the user in the environment and potentially other features such as fog and clouds that could also be generated using fractal techniques.

3 Background

The use of fractal landscapes has been rising in popularity in both the CGI film industry (Planetside, 2015) and the games industry, with some notable examples including the soon to be released “No Man’s Sky” (Parkin, 2015) and the older “Minecraft” (Young, 2015) which still maintains thousands of sales a day (Q4, 2015) with has over 21 million units sold on pc and mac alone (Mojang, 2015). The use of fractal landscapes in these games allows developers to generate infinite worlds or even galaxies; allowing players to explore these vast landscapes can clearly lead to a successful product, so as long as the landscape is not only huge, but interesting as well.

A fractal is a never ending pattern that is self-similar across different scales (Fractal Foundation, n.d.); to put it more simply this means that something is fractal if it has the same pattern as a whole as it does when zoomed in. The mathematician Mandelbrot was the first to notice that fractal patterns appear a lot in nature (Mandelbrot, 1982), such as in trees, rivers, mountains, and coastlines. A fractal landscape is natural looking terrain generated by an algorithm that makes use of a random variable to produce a surface that is somewhat fractal in nature.

3.1 Comparison of Technologies

3.1.1 Graphics APIs

In order to render the landscape a graphics API was required; the two that are the most widely used today are DirectX, a proprietary API created by Microsoft (Microsoft, n.d.), and OpenGL, an open source API (Kronos Group, n.d.). The following is a short comparison of each of these technologies.

DirectX is closed source and limited to Microsoft platforms (Windows, Xbox), although it is possible to enable it to run on other platforms through third party software such as W.I.N.E. (WINE HQ, n.d.), however some performance loss can be expected. Traditionally, DirectX is the first to implement new technologies, and as such it has been the standard for the games industry for many years although OpenGL is increasing in popularity.

OpenGL is an open source, cross platform API available on most platforms including windows, Linux, and mac (Kronos Group, n.d.). As it is open source there are a large number of tutorials available online, however many of them use deprecated versions of the API. At the GDC in 2014 several experts suggested that OpenGL could be up to fifteen times faster than DirectX (Walton, 2014), although such drastic performance differences are unlikely to be seen in the context of this project.

The version of each should also be considered, for instances using tessellation or geometry shaders limits compatibility to systems that support OpenGL version 4 (OpenGL, 2014) or DirectX version 11 (Wikipedia, 2016).

3.1.2 Programming Languages

The options for programming languages are somewhat restrained by the choice of graphics API and the authors familiarity with the languages. Although OpenGL is supported by a wide range of languages the author is only comfortable enough using the bindings for C++ or C#.

While C# is a more modern language and is technically easier to use, the greater performance speed of C++ and the authors deeper familiarity with using it in conjunction with OpenGL mean that it has been decided that C++ will be used for this project.

A few libraries will also be used to help with some of the more fundamental OpenGL and Maths functions. The SDL2 library is used to make creating a window and OpenGL context simple as well as to handle user input (SDL, 2016). The Maths library, GLM, has also been chosen as it was specifically creating for use with OpenGL (GLM, 2016). Both libraries work natively in C++ which will aid in performance.

3.1.3 Graphic User Interface

A user interface will be required to allow the user simple control over the terrain; this will mostly be done through the manipulation of variables used during the terrain generation. It should also allow the user to swap between multiple cameras giving different controllable views of the landscapes. The final necessary feature the UI should enable is to allow the user to load and save different terrains as well as generate new ones at the push of a button.

3.1.3.1 Qt Framework

The Qt frame work is a powerful toolkit for developing easy to use user interfaces in many different languages. The toolkits components are written natively in C++ and support OpenGL, proving to be a popular tool for a wide variety of games (Qt, 2016).

3.1.3.2 AntTweakBar

AntTweakBar is a lightweight native C++ library designed to enable programmers to create simple intuitive user interfaces. It allows C++ variables to be simply bound to graphical controls (AntTweakBar, n.d.); this lends itself naturally to this projects use case, allowing the programmer to bind the variables that control the look of the landscape to simple controls. It is also completely free to use and redistribute and has examples of integration with a number of OpenGL libraries as well as DirectX.

3.2 Comparison of Algorithms

3.2.1 Terrain Generation

There are several existing methods to generate fractal landscapes; a brief summary of some of these algorithms is listed below.

3.2.1.1 Height Maps

Most terrain generation algorithms output a height map of some sort, as such it is necessary to define what a height map is. A height map is a two dimensional set of values, representing heights. This is often stored as a greyscale image where heights vary from zero (black) to 255 (white), this range can be easily stretched or compressed or a colour image could be used to increase the range.

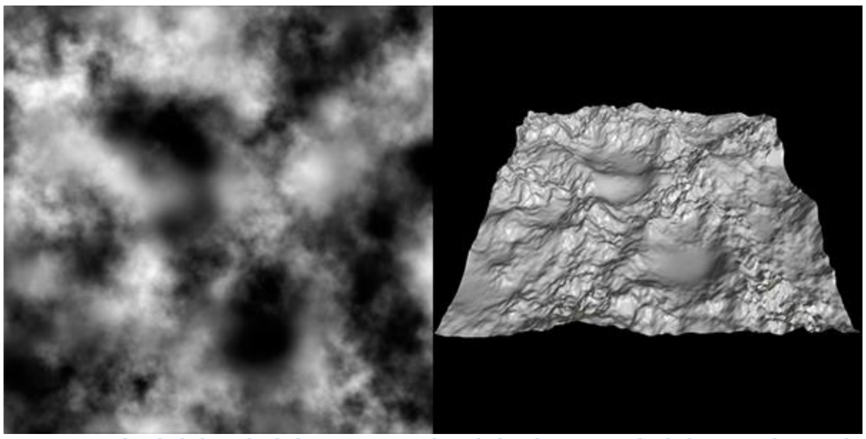


Figure 1 - On the left is a height map, on the right the same height map is rendered.

3.2.1.2 Mid-Point Displacement

One algorithm is mid-point displacement (Bird, Dickerson, George, 2013) in which a line or square is split at the mid-point and the height value of that point is displaced by a random value with in a set range. This process is repeated each time reducing the range to produce finer details until the desired level of detail is found. In figure 1 below you can see a 2D example of how this algorithm can quickly create a realistic landscape in just a few iterations.

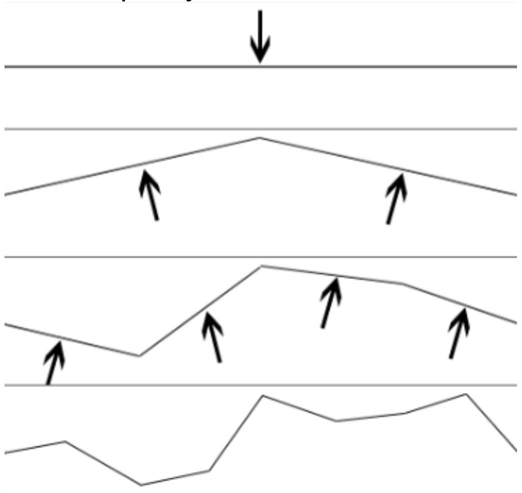


Figure 2 Mid-Point Displacement in 2D

3.2.1.3 Diamond Square Algorithm

The Diamond square algorithm builds on the previous algorithm, but attempts to remove the square artifacts that the terrain can exhibit by alternating the mid-point to be calculated between diamond and square patterns (Stranger, 2006). It is one of the most popular methods currently used.

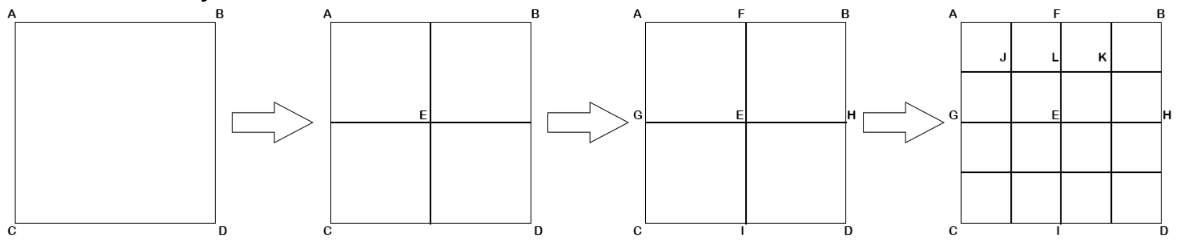


Figure 3 Diamond Square Algorithm from above

The calculation of the square midpoint is called the diamond step, and the calculation of the side midpoints is called the square step. Starting with a square of (ABCD), seeded with height values at the four corners, the first step is to calculate the height at the midpoint (E) by averaging the values at each corner (ABCD) and adding a random value based on the roughness of the terrain.
The next step is to calculate the midpoints of the line segments (F, G, H, and I), by averaging the corner values and midpoints of adjacent squares then once again adding a random value. To do this for the midpoint F, an average would be taken from A, E, and B before adding the random value. Then repeat square and diamond steps over and over again until a sufficient level of detail is acquired.

Both this method and the previous method have a couple of flaws, one being that real life landscapes aren't infinitely fractal in nature. Erosion and other natural processes cause landscapes to lose their self-similarity at the small scales (Bickford, 2012). The second flaw is that even if the processes are iterated enough times to provide a very high level of detail some artifacts will still be visible on the folds made by each iteration, causing the landscape to look more artificial.

3.2.1.4 Using Perlin Noise

One method that attempts to circumvent the flaws of the previous methods is to use a type of "noise" created by Ken Perlin, whilst he was working on the movie "Tron" (Perlin, 1999). At its core the idea is to think of landscapes as waveforms. With each mountain being made up of a large, low frequency wave and consecutively smaller and smaller waves each of which have higher and higher frequencies (figure 3).

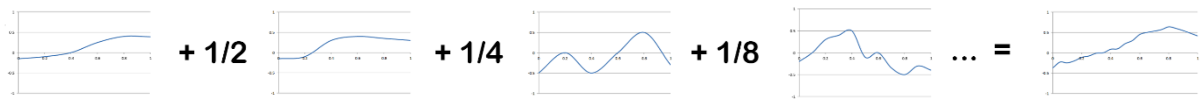


Figure 4 Graphs showing how a mountain scape can be created from lots of waves

In order to create a wave of Perlin noise, a list of random numbers must be created and then interpolated between using a non-linear interpolation function. It can then be extended into two or more dimensions by interpolating over the x-axis, then the y-axis, and so on.

An alternative method is to start with an image of random noise and either zoom in repeatedly into the same area or use multiple images and add the result of the zoom of each image together to create the height map in the same way as with the generated waves.

Perlin noise is a very popular technique throughout out the gaming and film industries; in fact, since its inception in 1983 Perlin noise has become an essential tool for generating surfaces, terrain, and natural appearing textures. In two or three dimensions it can be simply implemented to create textures and surfaces, and in the four dimensions to quickly animate them. Variances on the original algorithm, such as simplex noise (which removes some computational complexity), are used in almost every CGI tool to date and games like Minecraft (Persson, 2011) rely heavily on it for all their terrain generation.

3.2.1.5 Fast Fourier Transform

This technique also uses the idea that landscapes and waveforms are very similar in nature. It starts by generating a random white noise signal, and then applying a fast Fourier transform to the data (Bourke, 1997). This converts the data from the spatial domain, normal image space, into the frequency domain, which tells us how much of the signal lies within each frequency (ask a mathematician, 2012). This will basically look like slightly different white noise. So the next step is to filter the new data based on frequency, a number of different filters could be used, but one of the simplest ones is a "pink-noise" filter. In very simple terms this will remove a lot of the noise from the dataset, leaving it much smoother (LaBoiteaux, 2014). The final step is to transform the data back into the spatial domain using an inverse Fourier transform to do so.

The shape and scale of the landscape can be manipulated, by changing values in the filter that has been used and by changing the seed values when generating the original data. One of the more appealing aspects of this algorithm is that the surfaces that it creates tile perfectly, although this would perhaps be more useful when using the technique to create textures rather than terrains.

3.2.1.6 Particle Deposition

This technique is based on how some islands and mountain ranges are actually formed. In particular, it focuses on volcanic landscapes that are created by lava flow.
The basic principle is to drop particles onto an empty height map and simulate how they would naturally fall as the pile of particles that gradually increases (Shankel, 2000). An example of this technique can be seen in figure 4.

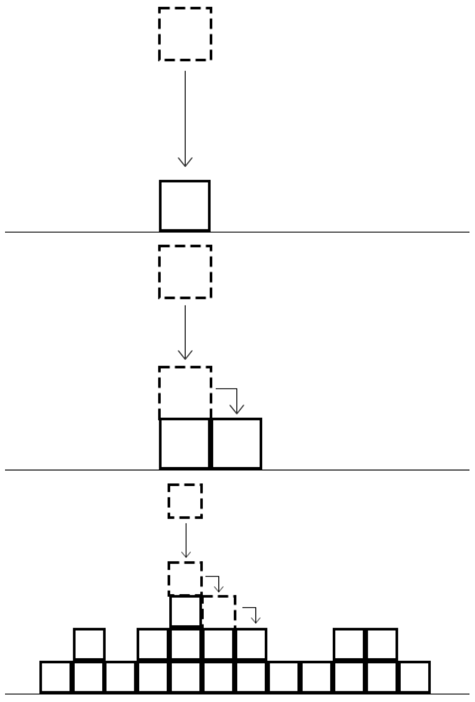


Figure 5 - Particle Displacement

Particles are dropped one at a time on top of each other, with each particle being pushed in each direction until it comes to rest. A particle is not at rest unless all of its immediate neighbours are equal or higher in altitude to it.
The dropping point can be varied randomly in both when it is moved and in which direction, to create a random landscape. Alternatively, the shape of the landscape can be relatively precisely manipulated by controlling how the particle drop point is moved.

3.2.1.7 User Control Over Terrain Generation

With the exception of the last technique discussed, most of these algorithms, at the most basic level, work in a somewhat similar manner. This similarity means that the methods of control that the user can have over the terrain generation are also similar.

Perhaps the property that will give the user the most drastic control over the landscapes is the seed value for the random number generator. Most of the techniques use a random number generator at some point, whether as seed values for the start of the grid in diamond squares or to create a source of noise in the Fourier transform technique. Saving the random numbers that are produced from the generator, also serves as a method for recreating the same landscape multiple times.

Another important value that the user could have control over is the “Roughness” value, this controls how steep the changes between points are and would have quite a large effect on how the landscapes looked overall.

The number of iterations or size of the grid of data can be changed to increase or decrease the density of the mesh providing large amount of detail or very basic meshes with very few details. If this number is too low however the landscape would not appear at all realistic, and if it is too high then it could cause performance issues.

The sea level of the created world could also be manipulated, while this would not affect the actual generated terrain it can change how the landscape is perceived. For example, a high sea level would mean only the highest mountains would be visible making it look like a world of islands, where as a low sea level would give vast continents and huge mountains.

3.2.1.7 Caves and Overhangs
One issue with a lot of these algorithms is that they have the inherent problem of not being able to model overhangs or caves (Bickford, 2012). Fortunately, there are a couple of ways around this, the first is to go over the vertex data after it has already been generated, and push or pull vertices to create caves and overhangs. The main problem with this issue is that it's very computationally expensive. The second option is that rather than treating the data produced as a height map; treat it as a density, where a value higher than or equal to zero represents the ground and a value lower than zero is air. To ensure that all the ground still starts at ground level and that there aren't floating pieces of ground, the data should be offset by the height. The downside to this technique is that it can produce some unsightly and unlikely landscapes.

3.2.2 Terrain Visualisation
Once generated the terrain needs to be rendered, below is a discussion of some of the methods to do this and various ways to optimise the performance of rendering terrain.

3.2.2.1 Standard Rendering
The most basic way of rendering a terrain is to simply assign one vertex per height value in the height map and assemble a triangle mesh from them, before sending them to the GPU. While simple and very easy to achieve, it is very expensive as a large number of polygons will need to be rendered. This would be somewhat acceptable if only a small section of terrain needed to be rendered, but as one of the aims of this project is to render an infinite amount of terrain it is unsuitable for this project.

However, some of the performance issues can be resolved by using LOD algorithms to dynamically change how many vertices are assigned to each height value reducing the load on the system for sections of terrain that require less detail. This could be achieved on the CPU or GPU and several algorithms to do this are discussed later.

Culling can also help to reduce the number of vertices that are passed to the GPU and reduce the load on the graphics card. There are two different types of culling, frustum and occlusion culling.

Frustum culling is where before the blocks of terrain are passed to the GPU, a calculation is done to determine whether the block is actually visible (inside the view frustum). If the block is not inside the view frustum then it is not passed to the GPU to be rendered, saving a large number of unnecessary draw calls (Lighthouse, 2015).

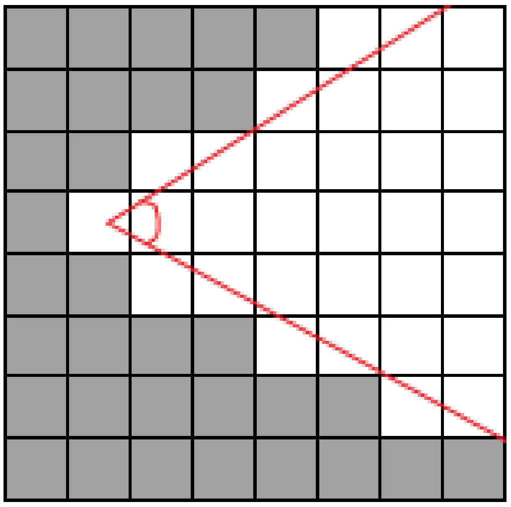


Figure 6 - Grey blocks of terrain are outside of frustum, so not rendered

3.2.2.2 Ray Casting

Ray Casting is a technique that replaces traditional rasterization and the z-buffer, by casting a ray for each pixel of the display in a straight line and checking for a collision at regular intervals, called steps, with a point of the terrain thus determining what colour each pixel should be (Abi-Chahla, 2009). A maximum distance is also set so that the ray does not carry on infinitely if no collision is found.

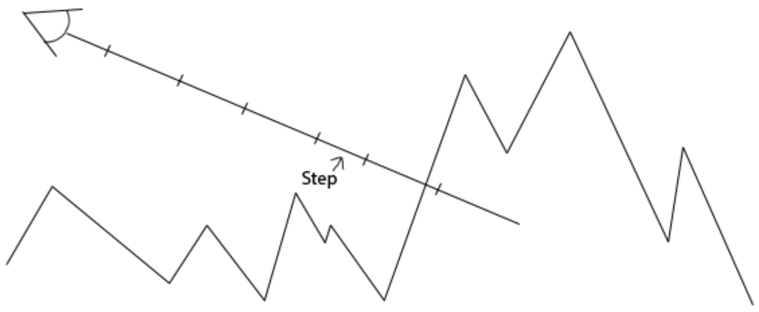


Figure 7 - A ray being fired through a terrain

The performance of this technique is dependent on the size of each interval, a larger size will be less computationally expensive but also less accurate so a balance must between accuracy and performance must be found.

3.2.3 Level of Detail Algorithms

Level of detail or LOD as it commonly known is an essential component for rendering landscapes in real time. The principle of it is to render the terrain that is further away in less detail than the terrain that is closer to the camera, therefore reducing the amount of work that the GPU has to do and improving performance.
As in real life people cannot see large quantities of detail on distant mountains, just the overall shape so it should be possible to implement level of detail without the user being aware of any change (Figure 4). It should be pointed out that level of detail is not just restricted to terrains, but used throughout all 3D graphics to reduce the overall workload (David Luebke, 2003).

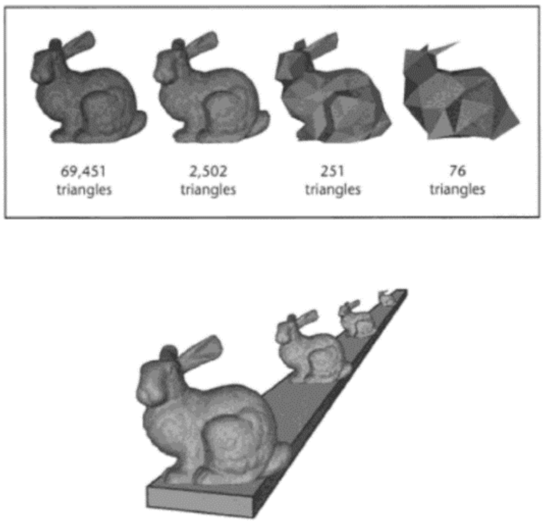


Figure 8 - Showing how the user cannot perceive the difference in a model made up of fewer triangles when it is far away (David Luebke, 2003)

3.2.3.1 ROAM Algorithm

ROAM stands for Real-time Optimally Adaptive Mesh, is one of the oldest and most common methods for applying LOD to terrain meshes. The algorithm uses two queues, sorted by priority. The first queue is called the “Drive Split Queue”, this stores a list of triangle splits making it simple to increase the LOD of the terrain by just repeatedly splitting the top triangle in the queue. The second queue is called the “Drive Merge Queue”, storing a list of triangle merge operations, so to decrease the level of detail of the terrain repeatedly execute the top operation in the queue (Mark Duchaineau, n.d.). The split-merge technique can be seen below in figure 6.

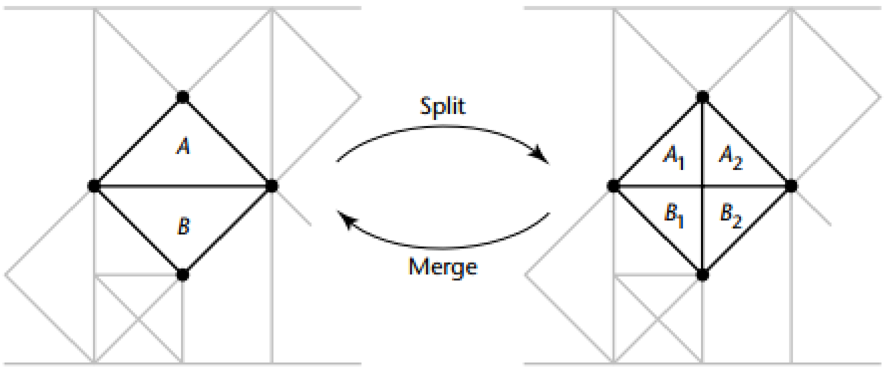


Figure 9 - Shows the split-merge technique for ROAM on a terrain map (David Luebke, 2003)

This algorithm works excellently from a purely visual perspective, however as it originates from before any major advances in shading technologies it is entirely CPU based meaning that unfortunately it can’t keep up, in terms of performance, with more modern techniques that make use of shaders.

3.2.3.2 Chunked LOD

Chunked LOD uses a quad-tree structure (Ulrich, 2000), to sub-divide a tile of terrain into consecutively smaller tiles based on the proximity of the player and a set parameter called the LOD threshold. Each sub-divided tile is made up of the same number of vertices but represents a half the area in world space for each sub-divide, this means that the detail will increase the closer the player gets to the terrain. Figure 7 shows the sub-divides closer to the player.

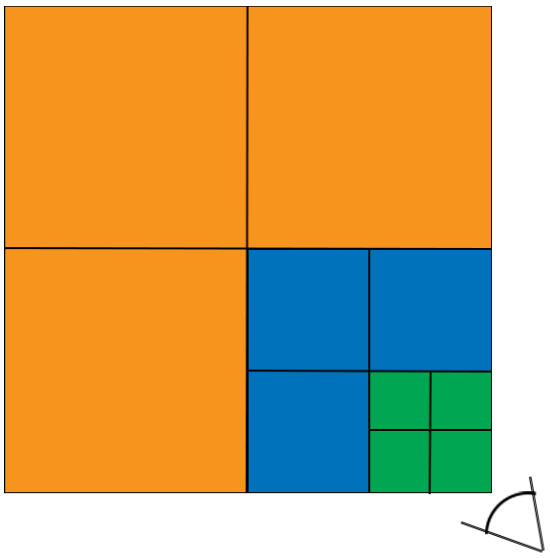


Figure 10 - Chunked LOD, each colour represents a sub-divide.

This technique should perform better than ROAM; however, it is possible that some terrain “popping” would occur as sub-divides are made.

3.2.3.3 Geometry Clip Maps
Using the geometry clip map technique the terrain is dynamically divided up into groups of different levels of detail depending on the distance from the camera, producing a concentric grid of different levels of detail that move with the camera. This eliminates the “pop” effect often seen in discrete level of detail algorithms like Chunked LOD (Pharr, 2005).

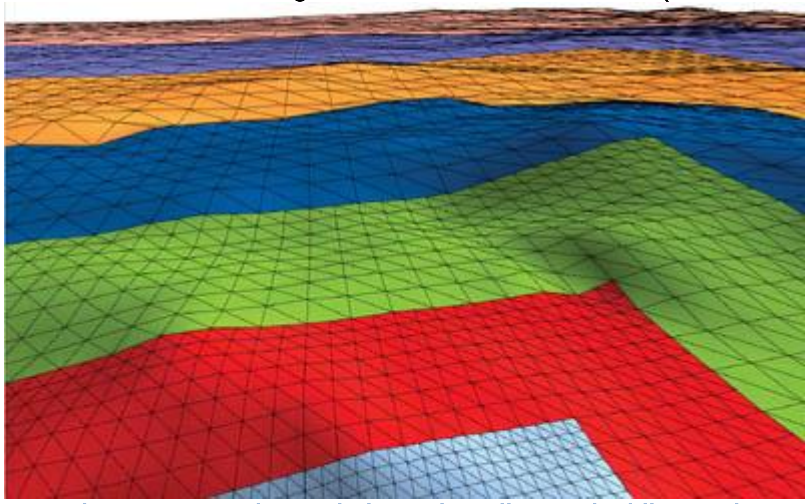


Figure 11 - Terrain Rendering Using Clip Map (nVidia, 2005)

3.2.3.4 Geometry Mip Maps
In this technique, the terrain is divided into patches of uniform size, and different levels of detail are generated depending on the distance the patch is from the camera. A low detail patch consisting of less vertices will be shown when the block is far away from the camera, and a patch consisting of more vertices will be swapped in as the block gets closer. This technique is simple to implement; however noticeable popping would be noticeable as blocks are swapped for ones of different detail.

3.2.3.5 Hardware Tessellation
A more recent innovation in level of detail techniques is using hardware tessellation is to use tessellation shaders that have been implemented in OpenGL 4 and DirectX 11 (Real-Time Rendering, 2008). They dynamically change the level of detail based on passed in tessellation parameters (Aviel, n.d.).

In OpenGL 4 the tessellation stage adds two new programmable shaders and a fixed function generator between them. The tessellation control shader takes the input of a simple patch of vertices and calculates how many times the patch should be sub-divided, this is defined by the “Inner Tessellation” level and the “Outer Tessellation” level as shown in figure 12. The fixed step of the tessellation stage then divides the patch up the appropriate amount of times and outputs a new set of vertices.

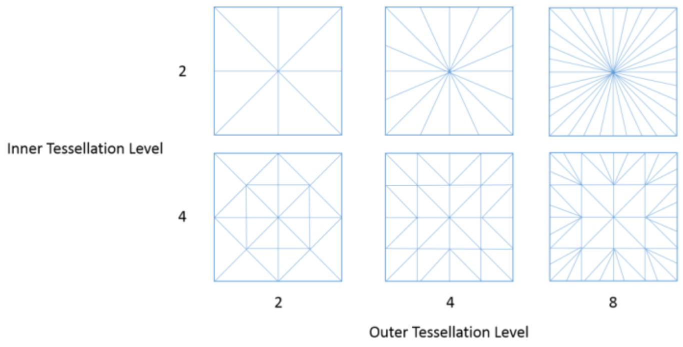


Figure 12 - How Tessellation Levels Work

The second programmable shader is the tessellation evaluation shader, it's job is to assign each vertex a position in world space and in the case of rendering terrain displace the height of the vertex using values from the generated height map.

The only real drawback to this implementation is that as it is relatively new it doesn't have as wide range of hardware support as the methods that were discussed earlier, despite this drawback tessellation is the chosen level of detail implementation for this project.

4 Technical Development

4.1 System Design

4.1.1 Technology

Following analysis from the research presented in the background chapter it was decided that this project should use the combination of C++ and OpenGL due to both their performance benefits and the authors previous experience developing with them. The SDL library has also been used to make window and context creation easier, as well as to handle user input. GLM was also chosen as a comprehensive math's library that has been specifically designed for OpenGL.

It has also been decided to use tessellation as the preferred level of detail method, as such the application will only run properly on systems that support OpenGL version four or higher. Fortunately, the target system meets this requirement.

The chosen algorithm for generating the terrain is the diamond-square method, this will generate the terrain on the CPU, so the application is multi-threaded in order to maintain acceptable performance.

4.1.2 Architecture

As this project is a real-time simulation it was logical to create a game framework to handle the constant update and render calls.

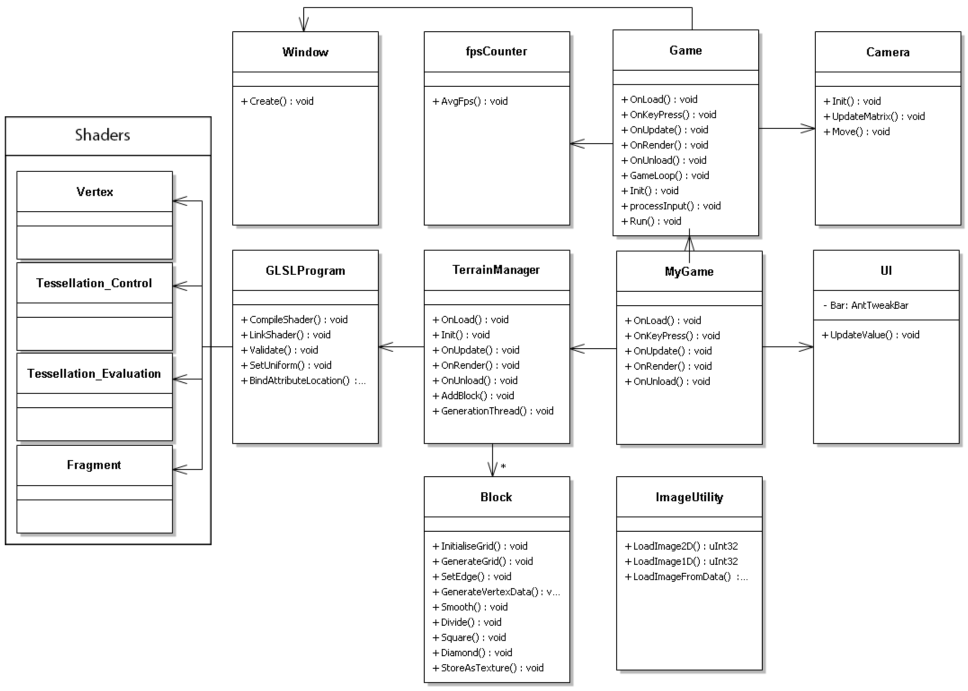


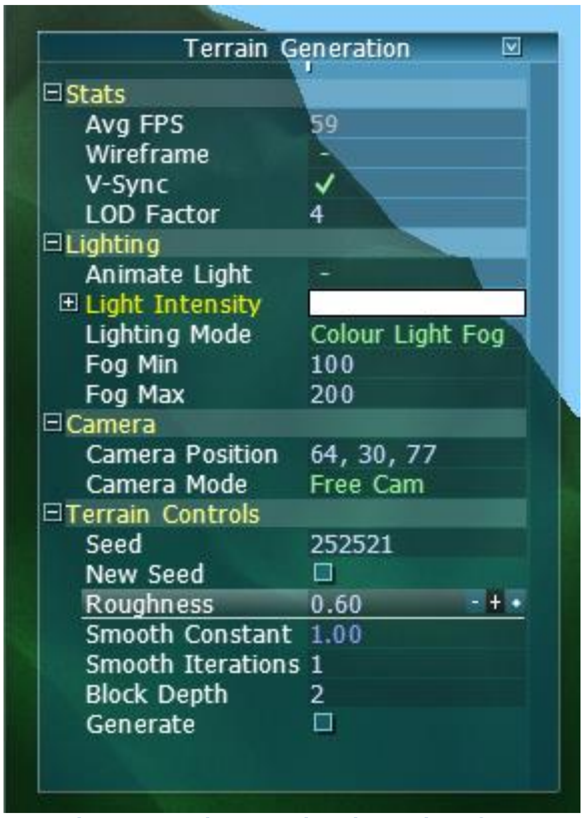
Figure 13 - Class Diagram of Completed System

Figure 13 shows a class diagram of the game framework as well as the classes created to handle the generation and rendering of the terrain, in order to avoid an overly complex diagram many methods and variables have been omitted.

Game Class – The game class provides a template for My Game. It holds a camera object, window object and an fps counter used to monitor performance. The OnLoad, OnUpdate, OnRender, OnUnload, and OnKeyPress methods are simple stubs designed to be overridden by the My Game class in a similar way to how game frameworks like XNA work. The Game class also holds the main game loop, which runs the OnUpdate, OnRender, and OnKeyPress methods repeatedly.
Window Class – The window class uses the SDL library to create a window and OpenGL context of a specified size. It also handles some graphics options like vSync and double buffering.
Camera Class – The camera class handles the creation of view and projection matrices, as well as helper methods to alter the camera's positon, pitch, and yaw.
Fps Counter Class – The fps counter records the time it takes for each frame to complete and outputs an average over a defined number of frames.
My Game Class – This class overloads the Game class, processing key inputs to move the camera, and calling the update and render methods for the Terrain Manager and UI objects that it holds.
UI Class – The UI class uses the AntTweakBar library to create a simple user interface to edit variable contained within the My Game and Terrain Manager classes.
Image Utility Class – This static utility class holds methods to generate 1D and 2D textures from image files using the SDL Image library, as well as the method to generate a 2D texture out of terrain height and normal information.
Terrain Manager Class – The terrain manager class handles the generation, rendering, and management of the terrain blocks. It generates the initial terrain blocks then creates new blocks, on the generation thread, as the user moves around the world, ensuring that they are properly stitched together with existing blocks. It also handles removing blocks that go out of render distance. The class also stores the necessary information to recreate the block of terrain if the user returns to it and methods for extracting data such as the height value from a block at the current camera position. It also controls the GLSL Program that is applied to the terrain, as wells as the creation of the vertex array and buffer objects.
Block – The block class is where the diamond square algorithm is implemented. Vertex data, such as the surface normal vectors, are also calculated in here from the generated height map before being loaded into a texture using the Image Utility class and rendered in the terrain manager class.
GLSL Program Class – The GLSL program class handles everything shader related. It is capable of compiling and linking every stage of the modern OpenGL pipeline, from passed in shader files. It also contains large amounts of error handling to output GLSL compiler errors, which is very useful for debugging shaders that were mostly written in a text editor without any syntax checking. There are also several helper methods to set uniform values in the shaders reducing the amount and complexity of code needed in the rest of the application.
Shaders – The vertex shader is incredibly simple, it's only function is to offset the initial vertices with values from the height map before passing them onto the tessellation control shader. The tessellation control shader determines the tessellation levels using the level of details algorithm that will be described in greater detail below. After the tessellation levels have been applied the evaluation shader sets the normal values and displaces the new vertices with information from the height map, before passing them to the fragment shader where colour, lighting, and fog are applied.

4.2 UI Design

A decision was made to use “AntTweakBar” for the user interface; this was mainly due to its lightweight nature and the fact that it was designed for almost the exact same use case as this project, to tweak variables in the terrain algorithms. AntTweakBar is built natively for C++ and is very easy to interface with, binding variables from My Game and Terrain Manager to UI components that enable the user to very simply tweak them and see the effect of doing so in real time.



Terrain Generation

⊟ Stats	
Avg FPS	59
Wireframe	-
V-Sync	✓
LOD Factor	4
⊟ Lighting	
Animate Light	-
⊞ Light Intensity	
Lighting Mode	Colour Light Fog
Fog Min	100
Fog Max	200
⊟ Camera	
Camera Position	64, 30, 77
Camera Mode	Free Cam
⊟ Terrain Controls	
Seed	252521
New Seed	☐
Roughness	0.60
Smooth Constant	1.00
Smooth Iterations	1
Block Depth	2
Generate	☐

Figure 14 - The completed user interface

The user interface appears in the top right of the screen, although it can be hidden or moved so that it does not obscure any part of the terrain. It is controlled using the mouse, by using clicking on the plus or minus buttons or by manually enter values. The user interface is divided up in to four sections, the first of which gives the user performance statistics and the ability to adjust some render and graphics settings. The user can choose to render the terrain as a wire frame or adjust the level of detail factor to control the tessellation levels. The second section gives the user the means to change all of the lighting variables and adjust the linear fog thresholds. The third section allows the user to switch camera mode between; flying, free cam, and top down view, as well as displaying the cameras position. The final section gives the user control over how the terrain is generated, they can change the seed value from which all blocks are based or adjust the roughness value used in the diamond-square algorithm, and apply smoothing to the generated height map. They can also choose to generate an entirely new terrain or adjust the number of blocks that surround the user at any one time.

4.3 System Implementation

This section of the report discusses how several major parts of this project were implemented, and the various problems and solutions associated with each of them. It is ordered chronologically.

4.3.1 Terrain Generation – The Diamond Square Algorithm

The diamond square algorithm was chosen to be used for terrain generation in this project, as a balance between complexity and realism. The implementation of the diamond square algorithm was almost identical to how it was presented early in the background section, using a recursive function to progressively subdivide a square setting the height of each point to the average of four previously generated points, depending on whether it is the diamond or square step, plus a random offset value that decreases in size each iteration (roughness value) to produce the finer details. Figure 15 shows this projects implementation of diamond-square building a detailed terrain.

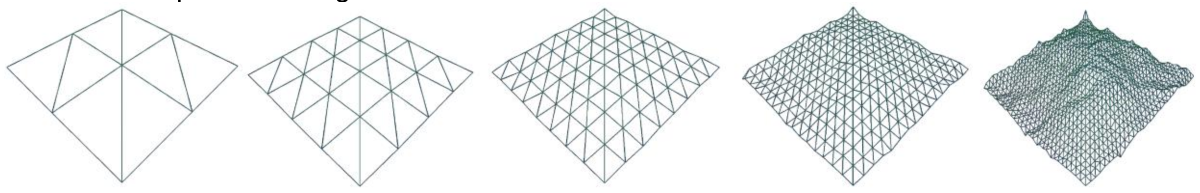


Figure 15 - Implementation of diamond-square algorithm

The user has control over the size of the generated height map, the seed used for the random number generator, as well as over the roughness value, between 0 and 1, which determines how mountainous or flat the generated terrain will be.

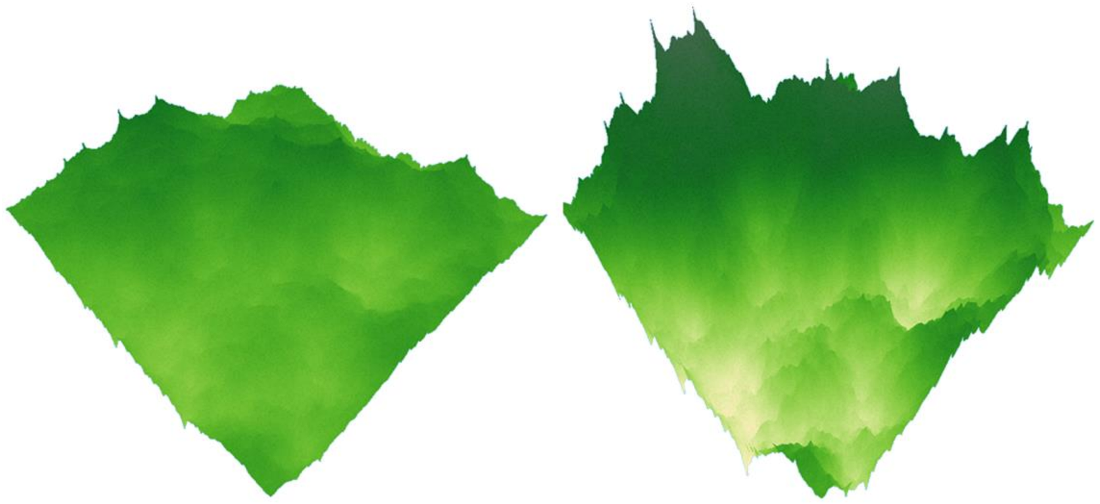


Figure 16 - Difference between roughness value of 0.25 (left) and 0.7 (right) for the same seed value

The algorithm outputs a two dimensional array of height values that can be used as a height map. In order to initially view the terrain, a simple mesh was created by creating a two dimensional array of vertices covering 0-128 across the x and z axis. The y value of these vertices was then set to the corresponding value in the generated height map, before generating a list of indices to create the correct triangle strip layout to draw.

As you can see from figure 16, the terrain the algorithm creates is suitably mountainous and a wide variety of different looking terrain blocks can be generated by altering the seed and roughness values. This implementation still has some flaws, such as the noticeable sharp peaks that are visible even at lower roughness values. An improvement on this implementation is discussed later in this section.

4.3.2 Texturing

In order for the terrain to look realistic it must to be coloured appropriately. An initial attempt to do this was done by hardcoding defined colour values for several height ranges in the vertex shader. And although this was an improvement on no variation in colour it leaves a lot to be desired (far left of figure 17).

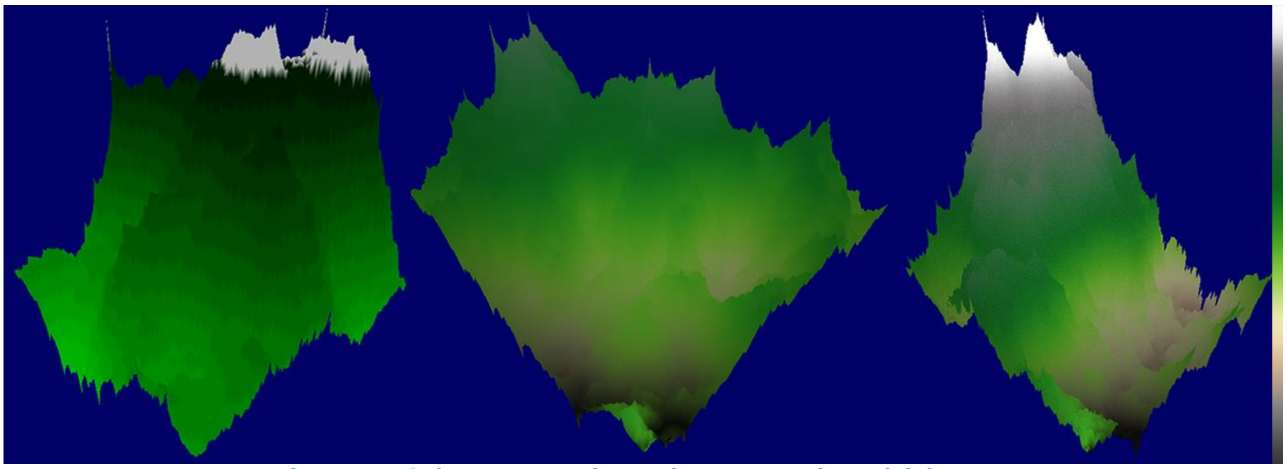


Figure 17 - Colour Progression and 1D Texture (Far Right)

The first attempt to improve this was to use a one dimensional texture as gradient rather than manually defining the colour for each range. A one dimensional texture is simply a texture that is one pixel wide, an example of the one used can be seen on the far right of figure 17 (it has been widened for visibility). Once loaded, using the SDL Image library, and passed into the shaders the fragment shader then assigns a colour value by sampling the 1D texture using the fragment height divided by the total height range for the block as the texture coordinate. The effect of this can be seen in on the centre block in figure 17. This is still not ideal and a final improvement was attempted by adding a pseudo random offset to the colour of each fragment irrespective of its height, this gives some additional variation across fragments allowing smaller details in the terrain to be more easily identified (figure 17 block on the right).

The final solution to texturing the terrain was deemed acceptable when combined with per fragment lighting. However more realistic and complex techniques could have been applied given more time, such as using multiple photo-realistic textures and blending them together across the terrain based on the height and gradient of the terrain.

4.3.3 Calculating Normals and Lighting

In order to properly light the terrain, the terrains normal vectors must be calculated. As the terrain is being generated on the CPU, it makes sense to generate the normals immediately after the height map has been generated.

There are a number of popular methods for generating normal vectors from a height map, the one implement in this project is as follows. Calculate the normal for each triangle in the map, using the cross-product of the triangles edges and add the normal to each point of the triangle. Once this has been done for every triangle the normal at each vertex is normalised.

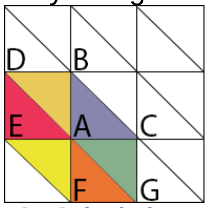


Figure 18 - Calculating Normals

To illustrate this further, if you wanted to calculate the normal of the vertex A in figure 18, you must first calculate the normals of every triangle that it is a part of (the coloured ones).

$$Normal\ of\ ABC = normalise\big(crossproduct(A - B, C - B)\big)$$

Then each normal is added to the normal vertex of A, before it is normalised giving you the final normal vector.

Once all the normal vertices have been calculated they can be passed along to the shaders, initial this was done using vertex attributes and a vertex array object, but later the normal and height value were stored in a two dimensional texture for the shader to sample.

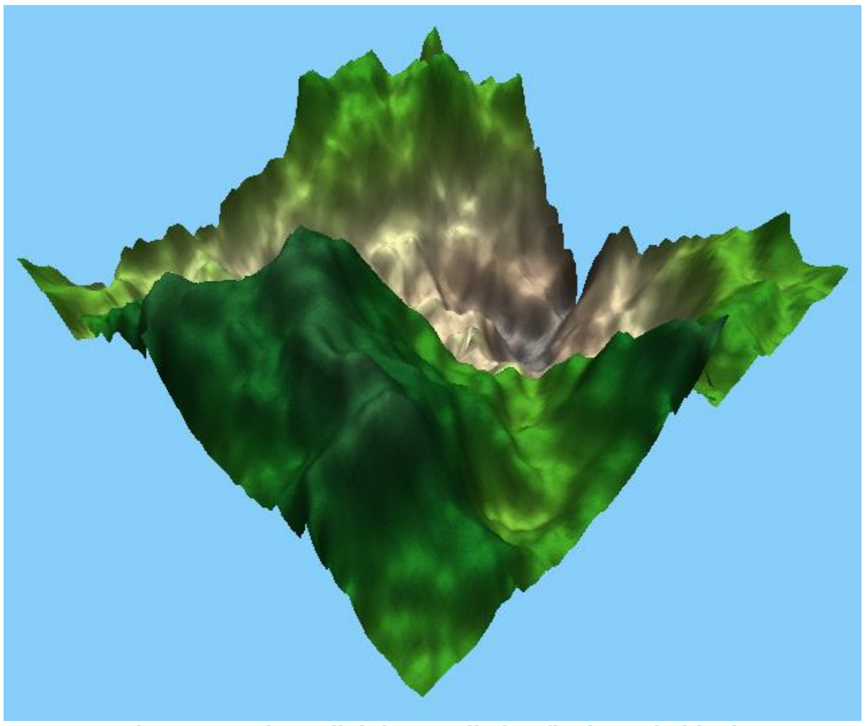


Figure 19 - Phong lighting applied to final terrain block

Once in the shader, the only lighting deemed necessary for rendering terrain was per-fragment Phong shading from directional source to simulate sun light (LightHouse, 2015). This was done in the standard manner, and as such will not be discussed any further from here.

4.3.4 Improving the Terrain Generation

At this point it was decided to try and improve the terrain generation algorithm. The steep peaks were the main issue that detracted from the realism of the terrain, unfortunately these are an artifact from the diamond-square algorithm itself and it would be overly complex to adapt the algorithm so that these were not generated. Instead it was decided to apply an optional smoothing filter to the generated height map, giving the user more control over the terrain generation.

A simple band smoothing filter was chosen (LightHouse, 2015), to be applied to the entire height map. The smoothing filter is applied to each vertex, and a new height is calculated based on the height of its immediate neighbours using the following equation.

$$height[x, z] = height[x - 1, z] * (1 - k) + height[x, z] * k$$

Where k is the smoothness constant between zero, completely flat terrain, and one, unchanged terrain.

In order for the smoothing filter to have optimum effect it must be applied in both directions for every row and column.

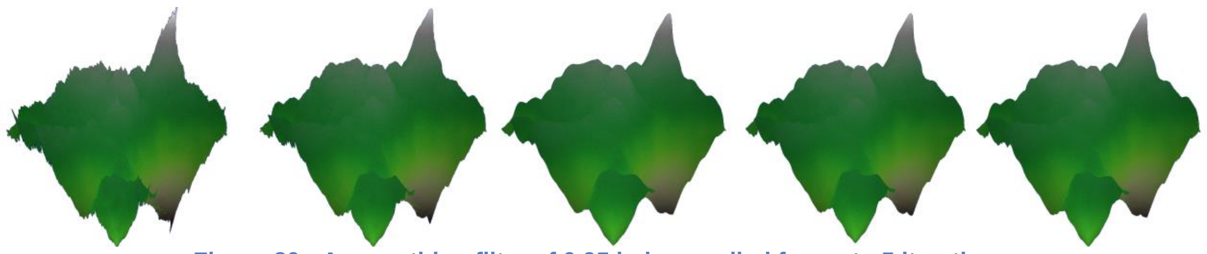


Figure 20 - A smoothing filter of 0.95 being applied for up to 5 iterations

Further control can be had over how smooth the terrain is by apply the filter several times, as can be seen in Figure 20. The drawback to this method is that it adds a lot of computation time to the terrain generation, depending on how many times the filter is applied, so a balance must be struck between performance and aesthetics.

4.3.5 Level of Detail - Tessellation

The general principle of tessellation has been explained earlier in the background section. As such this section will focus on the specific implementation of the level of detail algorithm and its limitations.

As the tessellation shaders will be sub-dividing patches, a patch must first be given created and drawn. In the terrain manager, a simple grid of nine vertices is defined in a vertex array object giving four squares that are 32 by 32 units in size. As the maximum tessellation factor can divide a terrain up to 64 times, the highest possible level of detail matches the dimensions of the height map for each block (128x128).

This also means that the height, and normal values must be passed to the shader in a new way. So instead of passing in the information as vertex attributes the height and normal are combined into a two dimensional texture where the first three values represent the normal and the fourth or alpha value represents the height.

The tessellation levels are calculated in the tessellation control shader. The level of detail calculation is made by determining the length of each of the squares edges in screen space and dividing by a user controlled factor (Wolff, 2011).

$$level\ of\ detail = \ clamp(distance(v0, v1)/lod\_factor, 1, 64)$$

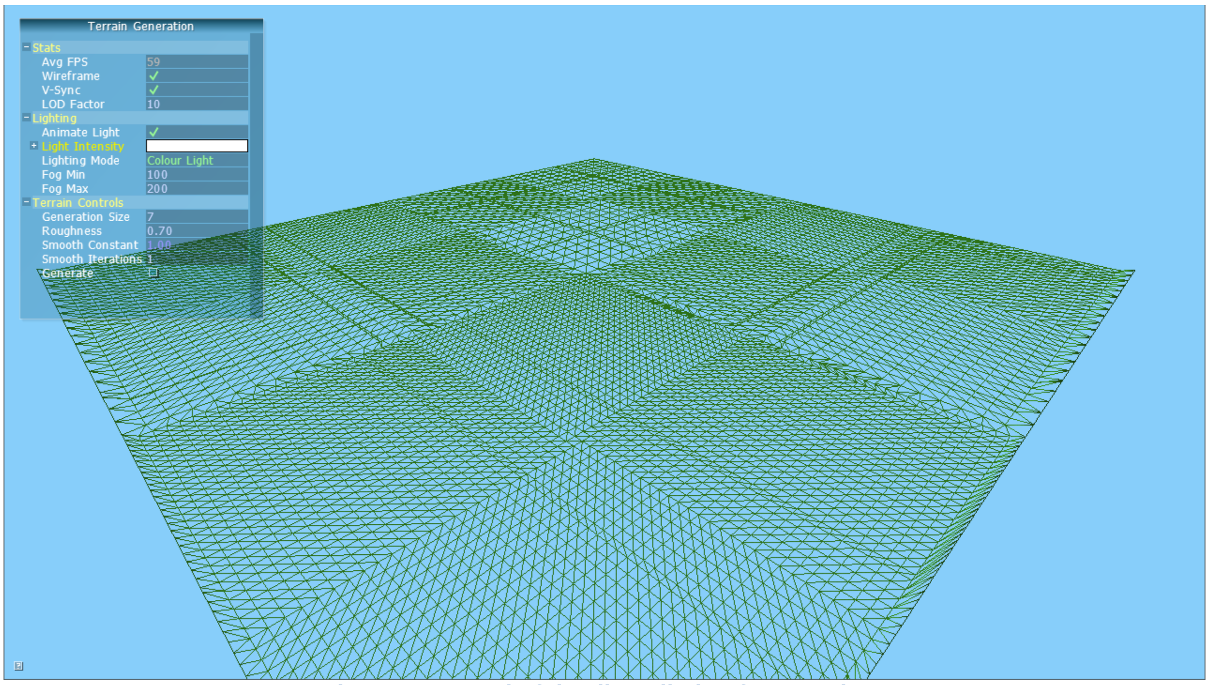
The length of the edge in screen space is determined by first multiplying each vertex by the model view projection matrix and then preforming the following calculation.

$$vertex\ in\ screen\ space = (clamp(vertex.xy, -1.3, 1.3) + 1)\ *\ (screen\_size * 0.5)$$

Where “screen_size” is a two dimensional vector containing the size of the screen in pixels.

Prior to this calculation the control shader determines if the patch is on screen, by testing each vertex against the view frustum, if the patch is not on screen then the tessellation levels are simply set to zero.

For quads there are six different tessellation levels that need to be set (Bush, 2015), one outer level for each edge and two inner values. The outer levels are set tl the level of detail calculation for the corresponding edge, and the inner levels are set to an average of the outer levels.

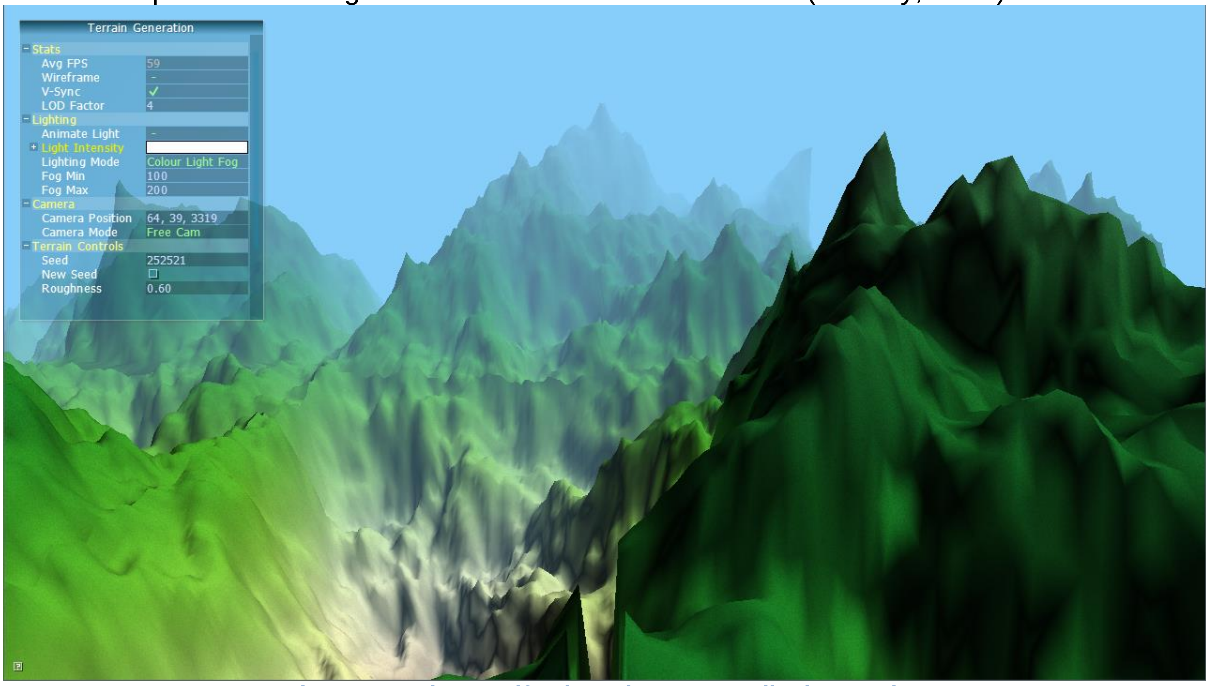


```
Terrain Generation
- Stats
    Avg FPS              59
    Wireframe            ✓
    V-Sync               ✓
    LOD Factor           10
- Lighting
    Animate Light        ✓
  + Light Intensity
    Lighting Mode        Colour Light
    Fog Min              100
    Fog Max              200
- Terrain Controls
    Generation Size      7
    Roughness            0.70
    Smooth Constant      1.00
    Smooth Iterations    1
    Generate             ☐
```

Figure 21 - Level of detail applied to four patches

Once all of the tessellation levels have been set, the fixed function “Tessellation Primitive Generator” creates the new points which are passed to the Tessellation Evaluation shader. The passed points are in a domain space, relative to their patch, so before anything useful can be done with them they must first be translated into normal space. Once in normal space the vertex height and normal can be extracted from the height and normal texture, before being passed onto the fragment shader for texturing and lighting.

While this approach looks good the majority of the time, one issue that has become apparent is that when the camera is perpendicular to the edge of a patch, some strange artifacting occurs. This could potentially be resolved, by creating a circle around the edge and testing its screen space radius to give a more accurate level of detail (Cantlay, 2011).



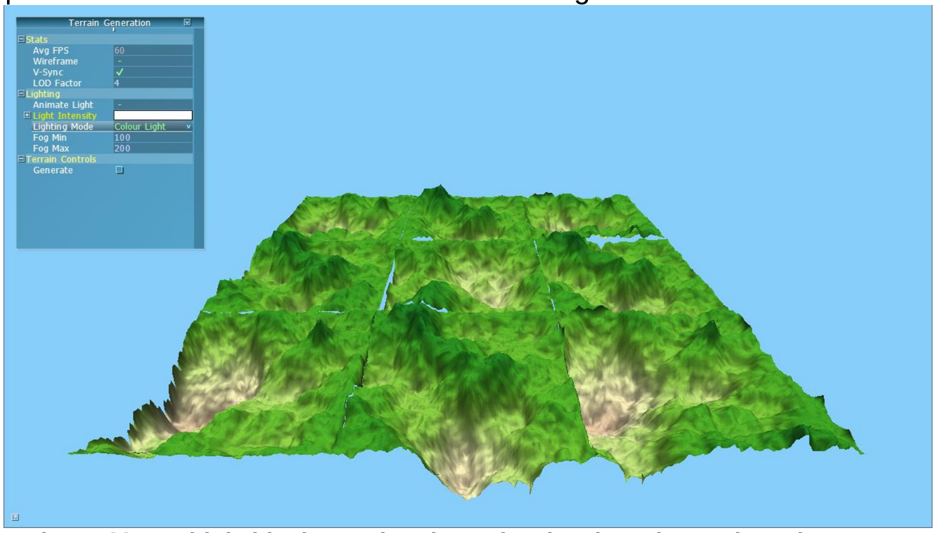
```
Terrain Generation
- Stats
    Avg FPS              59
    Wireframe            -
    V-Sync               ✓
    LOD Factor           4
- Lighting
    Animate Light        -
  + Light Intensity
    Lighting Mode        Colour Light Fog
    Fog Min              100
    Fog Max              200
- Camera
    Camera Position      64, 39, 3319
    Camera Mode          Free Cam
- Terrain Controls
    Seed                 252521
    New Seed             ☐
    Roughness            0.60
```

Figure 22 - Show artifacting when perpendicular to edge

4.3.6 Infinite Terrain

To create infinite terrain, the first obstacle that must be overcome is to draw multiple blocks of terrain. To do this each block is given an x, z position, which when added to the global seed in the terrain manager is also used for the blocks seed to generate its own height and normal map texture.

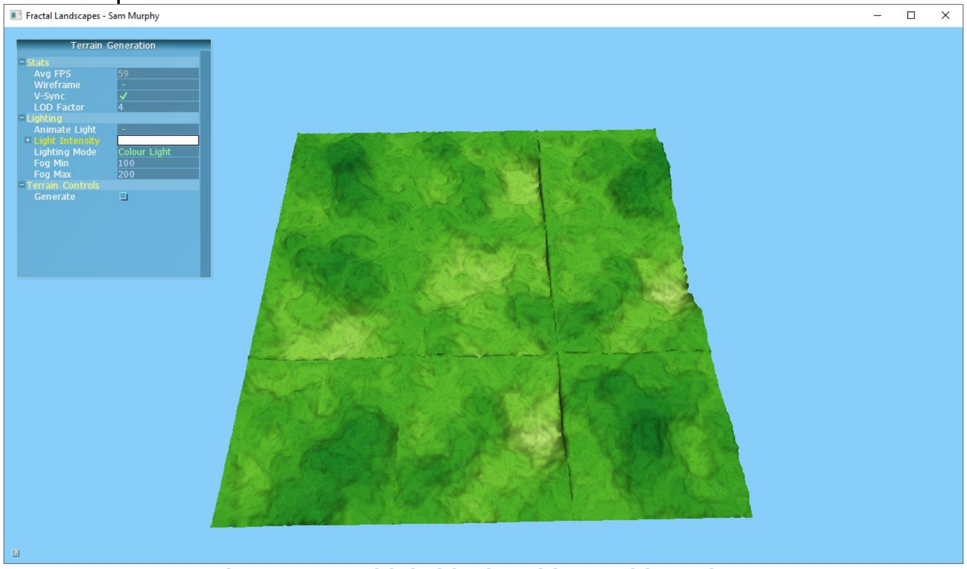
The terrain manager stores a list of all the generated blocks and in it's on render method it loops through them, setting the shaders model matrix to be a translation matrix created from the blocks x, z position and also setting the shaders sample2D to be the shaders height and normal map texture. The result of this can be seen in figure 23.



Avg FPS	60
Wireframe	-
V-Sync	✓
LOD Factor	4
Animate Light	-
Light Intensity	
Lighting Mode	Colour Light
Fog Min	100
Fog Max	200

Figure 23 - Multiple blocks rendered, cracks showing where edges don't meet

It is immediately apparent that the terrain looks wrong, the edges of the terrain do no match. In order to resolve this, the blocks of terrain must be generated in a specific order. The first block to be generated must be the centre block, where the camera starts. Once this block has been generated each block that shares an edge with it can then be created, but before the diamond-square algorithm can generate its height map, each edge of the height map that adjoins the centre block must be set to the values of the corresponding edge of the centre block. Once that is done the rest of the blocks data can be generated and then the process can be repeated for the corner blocks.



Avg FPS	59
Wireframe	-
V-Sync	✓
LOD Factor	4
Animate Light	-
Light Intensity	
Lighting Mode	Colour Light
Fog Min	100
Fog Max	200

Figure 24 - Multiple blocks with matching edges

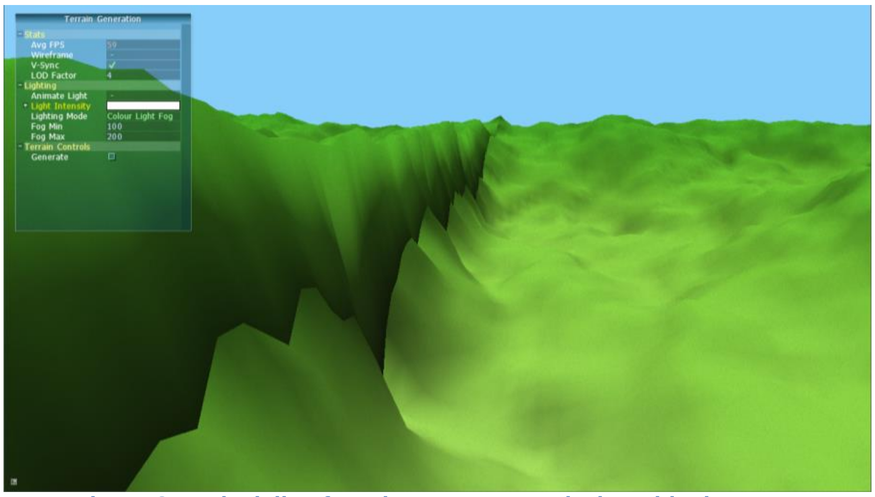
The next step to achieve infinite terrain is to generate blocks as the camera moves around the world. This is done by recording the cameras initial position as the last location that terrain was generated, then once the camera has moved more than half a block from that location the terrain manager will attempt to add all of the blocks that should be surrounding the current block. If the block it is trying to add already exist it is skipped and if not, then its edges are matched to existing blocks and its height map can be generated. The last step is to set the last location terrain was generated to the cameras new position, so that when the camera moves far enough again more terrain can be generated.

The terrain manager also checks when a block is more than the user defined block depth away from the camera, if it is the block is removed so that the application does not run out of memory. In order to maintain performance, the terrain generation is also done on a separate thread to the rest of the application logic and rendering code, if this was not the case then the user would be unable to do anything every time they were waiting for terrain to be generated.

The terrain generation thread is spawned when a new terrain manager is created, it runs continuously until the application is exited. It monitors the last position that terrain was generated and adds the new blocks when the camera has moved far enough. It also uses a mutex to lock the various lists it accesses so that no errors are caused by adding and removing terrain blocks. This implementation ensures that the performance of the application suffers little impact when generating terrain, so can still be considered real-time.

In order to ensure that when the user returns to a block that has been previously generated but then deleted, some information about each generated block must be stored for the duration of the time the application is running. This is necessary because although the seed value can be recreated from the blocks x, z position there is no guarantee that the adjacent blocks that were used to set the edge values still exist. As such for every block of terrain that has been previously generated, each of its edge values must be saved along with its x, z position that is used as a key to identify which block the edges belong to. Then when recreating the block, a check is performed to see if it has been previously generated and if so its edges are set to the ones that have been stored for it.

There were two main issues discovered when generating multiple blocks of terrain, the first of which was that the normals at the edge of each block were not smoothed with its neighbours this creates distinct black lines along some edges.



Avg FPS	59
Wireframe	-
V-Sync	✓
LOD Factor	4
Animate Light	-
Light Intensity	
Lighting Mode	Colour Light Fog
Fog Min	100
Fog Max	200
Generate	☐

Figure 25 - Black line from incorrect normal where blocks meet.

In order to solve this issue, it would be necessary to recalculate the normal for the edge vertices incorporating the triangles that each edge vertex is a part of in the adjacent block.

As this would require significant reworking of the code and normal generation algorithm, this solution has yet to be implement.

The second issue, is that when the smoothing function was applied to the terrain it would alter the height values at the edges of the blocks causing the reappearance of cracks in the terrain. This was resolved by re-writing the smoothing function so that it did not adjust the height values at the edges of the height map.

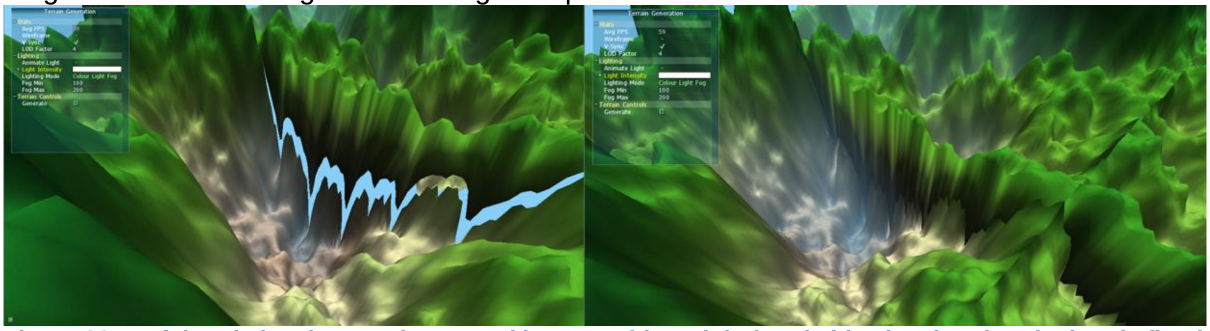


Figure 26 - Left hand showing cracks caused by smoothing, right hand side showing that the bug is fixed

4.3.7 Fog

In order to disguise the blocks of terrain appearing and disappearing as they are loaded in and deleted, it was decided to implement some fog so that it appeared the terrain was continuous. A simple linear fog was implemented (MbSoftworks, 2015) using the following equation.

$$fogFactor = clamp(\frac{(maxDistance - distance)}{(maxDistance - minDistance)}, 0, 1)$$

Where distance is the distance from the camera to the point on the terrain, and max and min distance are user defined parameters. The fog factor is then used to mix the sky colour with terrain colour, meaning that the fragments further away tend towards the colour of the sky giving the appearance of fog.

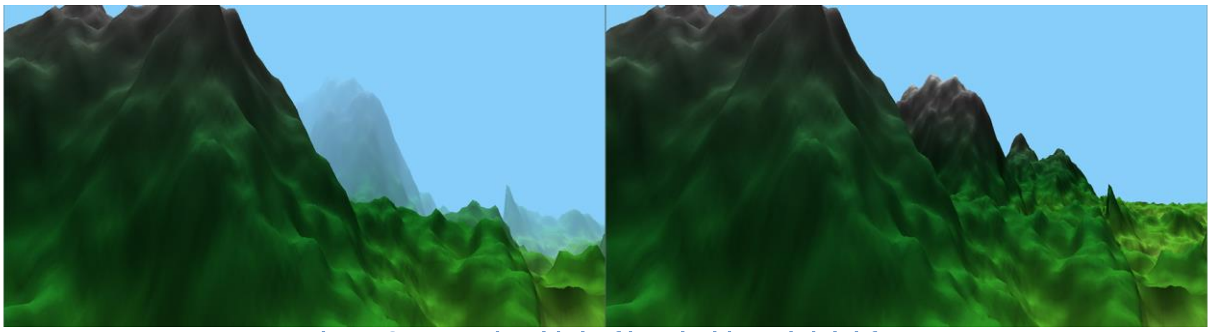


Figure 27 - Terrain with (Left) and without (Right) fog

Figure 27 illustrates the effect of the fog, on the left hand side the fog means that landscape smoothly blends into the sky, whereas on the right there is a jarring line where the terrain ends.

4.3.8 Flying Camera

Due to time constraints and poor planning no plane model was loaded, however a flying camera mode was added. This camera mode travels in the direction the camera is facing at a constant speed, while maintaining a defined distance off the ground. In order to achieve this a method of getting the height of the terrain in the position of the camera was added. This is done by taking the camera position, dividing it by the size of the block, and then multiplying the integer component of that by the block size to get the coordinate for the bottom left hand corner of the block. From this the blocks height map can be found, and using the floating point component of the earlier division the coordinates in block space can be calculated and the height extracted from the height map.

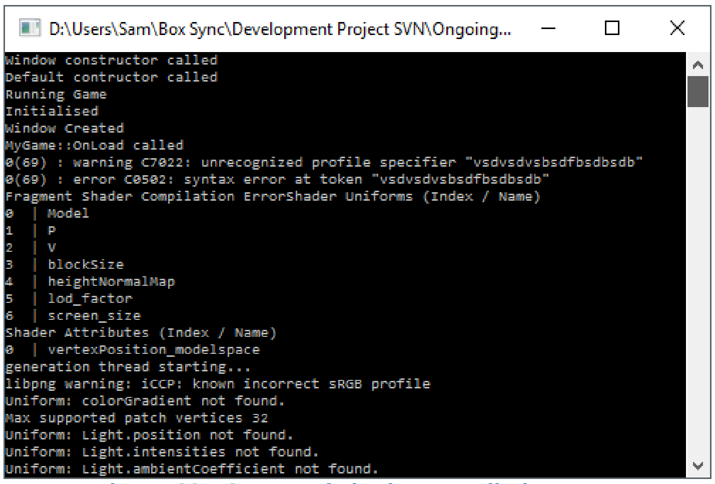
Once the height is calculated, the y position of the camera is simple offset by the height plus the defined amount. This technique for getting the height could also be used to implement terrain collisions, by extracting the normal and height from the height map although this has not been implemented in the current version of the project.

Two other camera modes are available; a free cam to move uninhibited around the terrain, and a top down camera used to test and demonstrate infinite block generation.

4.4 System Testing

4.4.1 Testing During Development

A large section of the project was created in GLSL. GLSL does not support any debugging beyond compiler errors, fortunately the GLSLProgram class that was created outputs these compilation errors along with other information in a user friendly manner allowing for quick bug fixing during the creation of the shaders.



```
Window constructor called
Default contructor called
Running Game
Initialised
Window Created
MyGame::OnLoad called
0(69) : warning C7022: unrecognized profile specifier "vsdvsdvsbsdfbsdbsdb"
0(69) : error C0502: syntax error at token "vsdvsdvsbsdfbsdbsdb"
Fragment Shader Compilation ErrorShader Uniforms (Index / Name)
0  | Model
1  | P
2  | V
3  | blockSize
4  | heightNormalMap
5  | lod_factor
6  | screen_size
Shader Attributes (Index / Name)
0  | vertexPosition_modelspace
generation thread starting...
libpng warning: iCCP: known incorrect sRGB profile
Uniform: colorGradient not found.
Max supported patch vertices 32
Uniform: Light.position not found.
Uniform: Light.intensities not found.
Uniform: Light.ambientCoefficient not found.
```

Figure 28 - Output of shader compilation error

The rest of the project developed in C++ using visual studio. Visual studio provides a lot of help for debugging and testing, however attempts were made to stick to strict coding practices to ensure that bugs were easy to identify and fix.

4.4.2 Performance and Stability Testing

To test both the performance and the stability of the application, the finished software was left to run in flying mode on multiple systems that met the minimum OpenGL requirements. The performance was evaluated by recording the average frames per second across this time. A table of the recorded values is below.

Test System	**Average Frames per Second**	
	With vSync	**Without vSync**
Test System A	60	317
Test System B	60	352
Test System C	60	228

5 Evaluation

In this section the finished project shall be reviewed to assess its success, failures and potential for further development.

5.1 Project Achievement

To assess whether or not the project can be deemed a success, it will be compared to the original goals laid out in the aims and objectives section earlier in the report.

Objective 1 – Generate 3D Terrain in Real-Time

It was stated that to complete this objective the generated terrain must be realistic, run in real-time, textured using shaders, and be infinite.
This objective was an almost complete success, the application generates terrain in real-time running at over 60 fps on the test systems, more than enough to be considered real-time. The terrain is textured appropriately using a 1D texture in the fragment shader. The only limitation on the size of the terrain is the maximum value of the floating points used for defining where the blocks of terrain are placed, so the terrain can be considered infinite.
The only negative in this section is that although the overall shape of the terrain and its level of detail can be considered realistic, the fractal algorithm at the applications core could use some improvements to make the terrain even more realistic. The issues with the normals at the joins between terrain chucks spoil the landscape, and more realistic textures could be used and blended based on both height and the gradient of the landscapes. Techniques could also be used to simulate erosion to produce more natural looking valleys.

Objective 2 – Simple Simulation of Flying Plane

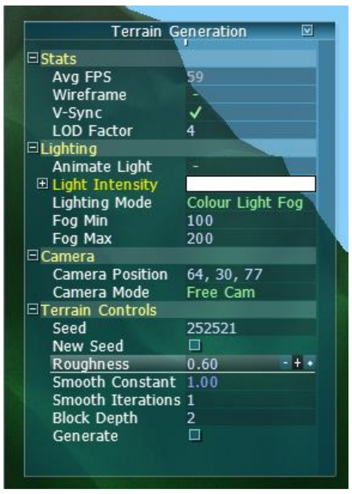
As no model of a plane was loaded due to time constraints and poor time management, this objective cannot be considered a success. However, the logic for the flying camera is in place and functional and all that would need to be implemented is the actual loading and rendering of the plane model for this objective to be completed, so it cannot be considered a complete failure.

Objective 3 – User Control of Terrain

All parts of this objective were completed, the user has control over a number of variables to affect how the terrain is generated and can generate an entirely new terrain based on any seed they like at the click of a button.

Objective 4 – Graphic User Interface

This object was to be considered met if the user interface allowed the user to control the terrain, as well as see performance statistics. As can be seen in figure 29, these conditions have been met. The user can control multiple terrain generation parameters as well as lighting and camera values. The user interface also outputs the average frames per second, which meets the requirement for performance statistics.



```
Terrain Generation
⊟ Stats
    Avg FPS            59
    Wireframe          -
    V-Sync             ✓
    LOD Factor         4
⊟ Lighting
    Animate Light      -
  ⊞ Light Intensity
    Lighting Mode      Colour Light Fog
    Fog Min            100
    Fog Max            200
⊟ Camera
    Camera Position    64, 30, 77
    Camera Mode        Free Cam
⊟ Terrain Controls
    Seed               252521
    New Seed           ☐
    Roughness          0.60
    Smooth Constant    1.00
    Smooth Iterations  1
    Block Depth        2
    Generate           ☐
```

Figure 29 - UI

Objective 5 – Additional Terrain Details

This objective can only be considered a partial success, due to the lack of the creation of any water or vegetation. However, some atmospheric details were added in the form of linear fog, used to help obscure the popping effect of additional terrain blocks being loaded. The additional details were omitted from this project, as there was not enough time to implement them and they were determined to be secondary objectives to the generation of fractal terrain.

Overall

As the majority of the primary objectives of this project have been met, this project is considered to be successful. However, it was hoped that more of the secondary aims could have been completed but due to time limitations that were mostly the result of poor time management some aspects were left unfinished.

5.2 Further Development

5.2.1 Texturing

Although the current texturing is acceptable, further work could be done to improve the appearance of the terrain. For instance, using multiple textures such as; a grass, rock, snow, and sand texture and blending them together using alpha blending based on not just the height of the terrain but also how steep or gentle the incline. This would mean that it did not appear as if grass grew or snow settled on steep cliffs as it currently does, but instead the slopes would be rocky while the plains would be grassy lending a more realistic appearance to the generated world.

5.2.2 Biomes and Vegetation

Biomes could be implemented, so that terrain blocks that are near each other are grouped sharing similar properties, such as the roughness constant and perhaps a set of biome dependent textures. This could create the appearance of smooth desert regions using sand textures and a low roughness value, and mountainous regions could be created using a high roughness value and rocky textures.

This could be expanded further by adding biome dependant vegetation, so that trees grow in low grassy areas, while the desert is sparsely populated by cacti.

5.2.3 Level of Detail

The implemented level of detail algorithm functions well. However as was discussed in early sections the current algorithm suffers from some bugs, these should be resolved using the techniques discussed in order to reducing artifacting across the terrain. In addition to this the current technique uses blocks of uniform size, performance could be increased by using a quad tree or similar structure to allow for blocks of non-uniform size to be created. As in the current implementation, a block that is very far away from the camera may only be being tessellated a tiny amount, allowing blocks that are further away to cover a larger area while being made up of less vertices would resolve this issue.

5.2.4 Water

Realistic water should be added, this should be done at the most basic level by adding a flat plane at sea level and the apply a custom water shader to it, to simulate the ocean. More complexity could be added by using erosion algorithms to simulate how the landscape has been errored by water over millennia and add rives along the paths defined by the created valleys.

5.2.5 Rendering Optimisations

Although the application has performed well across all three test systems, no block level frustum or occlusion culling is done, although patch level frustum culling is done in the tessellation control shader. Both of these techniques should be implemented to prevent unnecessary draw calls and improve overall performance.

6 Conclusion

This project started with the aim to create an infinite fractal world, generating terrain in real-time and allowing the user to fly through and experience that world, as well as giving them some control over how that world is generated. Based on the evaluation the project has certainly achieved that goal. With better time and project management more of the aims and objectives could have been achieved. However, the project also produced a reusable C++ and OpenGL game framework, and helped the author to gain a deeper understanding of C++, modern OpenGL shading techniques, such as tessellation, and the mathematics associated with games and 3D rendering.

Appendix A: Original Task List

#	Task Name	Description	Duration (days)
1	Initial Research	Research technologies, algorithms and context of the project	21
2	Initial Report	Write the initial report deliverable	14
3	Render Simple Mesh	Initial experimentation with graphics API’s to render a simple mesh	7
4	Research fractal algorithms	Research and experimentation with the different fractal algorithms for generating terrain.	14
5	Implement Infinite Scrolling World	Using the chosen algorithm, implement it so that the generated world seems infinite.	14
6	Optimization	Improve the implementation of the generation process and apply optimizations such as frustum culling around the user.	7
7	Load & Render Plane	Find a model of a plane, load into the program and render it.	7
8	Plane Controls	Enable the user to control the altitude and direction of the plane.	7
9	Implement UI	Design and implement a basic user interface.	7
10	Edit Terrain Settings	Enable the user to edit basic terrain settings through the user interface.	7
11	Performance Statistics	Output performance statistics to the user interface.	7
12	Interim Report	Write the interim report deliverable	21
13	Research Water	Research and experiment with different methods of rendering water.	14
14	Implement Lakes & Oceans	Using to chosen method, add randomly generated lakes and oceans to the world.	14
15	Research Foliage	Research and experiment with different ways of generating trees and other foliage.	14
16	Implement Foliage	Using the chosen methods add foliage to the world.	14
17	Continuous Testing	Testing the software throughout the development phase.	140
18	Final Testing	Testing the software to ensure stability and that it meets the set aims and objectives.	21
18	Final Report	Write the final report deliverable	42

Appendix B: Original Time Plan

		University Calendar Weeks																																
#	**Task Name**	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
1	Initial Research	▨	▨	▨																														
2	Initial report			▨	D																													
3	Render Simple Mesh					▨																												
4	Research fractal algorithms						▨	▨																										
5	Implement Infinite Scrolling World								▨	▨																								
6	Optimization										▨																							
7	Load & Render Plane											▨																						
8	Plane Controls												▨																					
9	Implement UI													▨																				
10	Edit Terrain Settings														▨																			
11	Performance Statistics															▨																		
12	Interim Report														▨	▨	D																	
13	Research Water																	▨	▨															
14	Implement Lakes & Oceans																			▨	▨													
15	Research Foliage																					▨	▨											
16	Implement Foliage																							▨	▨									
17	Continuous Testing					▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨									
18	Final Testing																									▨	▨	▨						
19	Final Report																												▨	▨	▨	▨	▨	D

Appendix C: Interim Time Plan

		University Calendar Weeks																																
#	**Task Name**	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
1	Initial Research	🟩	🟩	🟩																														
2	Initial report			🟩	🟩 D																													
3	Render Simple Mesh					🟩	🟩	🟩																										
4	Research fractal algorithms						🟩	🟩	🟩	🟩	🟩						▨	▨	▨															
12	Interim Report														🟩	🟩	🟩 D																	
5	Implement Infinite Scrolling World																	▨	▨	▨														
	Interim Demo																			🟥 D														
6	Optimization																				▨													
7	Load & Render Plane																					▨												
8	Plane Controls																					▨	▨											
9	Implement UI																							▨										
10	Edit Terrain Settings																							▨	▨									
11	Performance Statistics																							▨	▨									
13	Research Water																									▨	▨							
14	Implement Lakes & Oceans																										▨	▨						
15	Research Foliage																											▨	▨					
16	Implement Foliage																												▨					
17	Continuous Testing					🟩	🟩	🟩	🟩	🟩	🟩	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨					
18	Final Testing																												▨	▨	▨			
19	Final Report																												▨	▨	▨	▨	▨	🟥 D

Appendix D: Final Time Plan

		University Calendar Weeks																																
#	**Task Name**	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
1	Initial Research	🟩	🟩	🟩																														
2	Initial report			🟩	🟩 D																													
3	Render Simple Mesh					🟩	🟩	🟩																										
4	Research fractal algorithms						🟩	🟩	🟩	🟩	🟩																							
12	Interim Report														🟩	🟩	🟩 D																	
5	Edit Terrain Settings																	🟩	🟩	🟩														
	Interim Demo																			🟥 D														
8	Plane Controls																				🟩													
9	Implement UI																					🟩	🟩											
10	Implement Infinite Scrolling World																							🟩	🟩	🟩	🟩							
11	Performance Statistics																											🟩						
6	Optimization																												🟩					
17	Continuous Testing					🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩					
18	Final Testing																														🟩			
19	Final Report																															🟩	🟩	🟥 D

Appendix E: Risk Analysis

Risk	**Severity (L/M/H)**	**Likelihood (L/M/H)**	**Significance (Sev. x Like.)**	**How to Avoid**	**How to Recover**
Hard Drive Failure	*H*	*M*	*HM*	*Keep Backups*	*Reinstate from backups*
Data Loss	*H*	*L*	*LH*	*Use SVN*	*Restore from SVN*
Running out of Time	*H*	*L*	*HL*	*Maintain a detailed time plan, prioritizing the most important features*	*Adjust the time play and focus on the major features, cutting any unnecessary features.*
Chosen Technologies / Libraries Not working well together	*H*	*L*	*HL*	*Early prototyping with technologies / libraries to ensure compatibility and identify alternatives*	*Use alternatives instead.*
Lack of Experience	*M*	*L*	*ML*	*Do lots of research early on and try to use technologies that the developer is familiar with*	*Switch to technologies that the developer is more familiar with*

Appendix F: Test System Specifications

	System A	System B	System C
CPU	i7-5820k	i5	Phenom x4 – 955
GPU	980ti	780	GTX-670
RAM	16GB	8GB	8GB
Operating System	Windows 10	Windows 10	Windows 7

References

Abi-Chahla, F., 2009. *When Will Ray Tracing Replace Rasterization.* [Online]
Available at: http://www.tomshardware.co.uk/ray-tracing-rasterization,review-31636-2.html
[Accessed 04 05 2016].

AntTweakBar, n.d. *What is AntTweakBar.* [Online]
Available at: http://anttweakbar.sourceforge.net/doc/
[Accessed 20 01 2016].

ask a mathematician, 2012. *What is a Fourier transform? What is it used for?.* [Online]
Available at: http://www.askamathematician.com/2012/09/q-what-is-a-fourier-transform-what-is-it-used-for/
[Accessed 01 20 2016].

Aviel, n.d. *Why Hardware Tessellation is Awesome.* [Online]
Available at: http://www.nerdparadise.com/tech/graphics/hardwaretessellation/
[Accessed 20 01 2016].

Bickford, N., 2012. *Creating Fake Landscapes.* [Online]
Available at: https://nbickford.wordpress.com/2012/12/21/creating-fake-landscapes/
[Accessed 20 01 2016].

Bird, Dickerson, George, 2013. *Techiques for Fractal Terrain Generation.* [Online]
Available at:
https://web.williams.edu/Mathematics/sjmiller/public_html/hudson/Dickerson_Terrain.pdf
[Accessed 13 10 2015].

Bourke, P., 1997. *Frequency Synthesis of Landscapes (and clouds).* [Online]
Available at: http://paulbourke.net/fractals/noise/index.html
[Accessed 20 01 2016].

Bush, V., 2015. *Tessellated Terrain.* [Online]
Available at: http://victorbush.com/2015/01/tessellated-terrain/
[Accessed 04 05 2016].

Cantlay, I., 2011. *DirectX 11 Terrain Tessellation, nVidia,* s.l.: nVidia.

David Luebke, e. a., 2003. *Level of Detail for 3D Graphics.* 1st ed. San Fransico, California: Morgan Kaufmann Publishers.

GLM, 2016. *OpenGL Mathematics.* [Online]
Available at: http://glm.g-truc.net/0.9.7/index.html
[Accessed 04 05 2016].

Kronos Group, n.d. *OpenGL Overview.* [Online]
Available at: https://www.opengl.org/about/#1
[Accessed 13 10 2015].

LaBoiteaux, C., 2014. *Fast Fourier Terrain Generation.* [Online]
Available at: http://www.giantbomb.com/profile/bushpusherr/blog/graphics-blog-fast-fourier-terrain-generation/105554/
[Accessed 01 20 2016].

LightHouse, 2015. *Simulating Lighting Computations.* [Online]
Available at: http://www.lighthouse3d.com/opengl/terrain/index.php?light
[Accessed 04 05 2016].

LightHouse, 2015. *Smoothing.* [Online]
Available at: http://www.lighthouse3d.com/opengl/terrain/index.php?smoothing
[Accessed 04 05 2016].

Lighthouse, 2015. *View Frustum Culling.* [Online]
Available at: http://www.lighthouse3d.com/tutorials/view-frustum-culling/
[Accessed 04 05 2016].

Mark Duchaineau, e. a., n.d. *ROAMing Terrain: Real-tim Optimally Adapting Meshes.* [Online]
Available at: https://graphics.llnl.gov/ROAM/roam.pdf
[Accessed 01 20 2016].

MbSoftworks, 2015. *Fog Outside.* [Online]
Available at: http://www.mbsoftworks.sk/index.php?page=tutorials&series=1&tutorial=15
[Accessed 04 05 2016].

Microsoft, n.d. *Getting Started with Direct3D.* [Online]
Available at: https://msdn.microsoft.com/en-us/library/windows/desktop/hh769064(v=vs.85).aspx
[Accessed 13 10 2015].

Mojang, 2015. *Minecraft Statistics.* [Online]
Available at: https://minecraft.net/stats
[Accessed 13 10 2015].

nVidia, 2005. *Terrain Rendering Using GPU-Based Geometry Clipmaps.* [Online]
Available at: http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter02.html
[Accessed 04 05 2016].

OpenGL, 2014. *History of OpenGL.* [Online]
Available at: https://www.opengl.org/wiki/History_of_OpenGL
[Accessed 04 05 2016].

Parkin, S., 2015. *No Man’s Sky: the game where you can explore 18 quintillion planets.* s.l.:s.n.

Perlin, K., 1999. *Making Noise (A talk by Ken Perlin).* [Online]
Available at: http://www.noisemachine.com/talk1/4.html
[Accessed 20 01 2016].

Persson, M., 2011. *Terrain Generation, Part 1.* [Online]
Available at: http://notch.tumblr.com/post/3746989361/terrain-generation-part-1
[Accessed 20 01 2016].

Pharr, M., 2005. *GPU Gems 2.* s.l.:nVidia.

Planetside, S., 2015. *Terragen in Film.* [Online]
Available at: http://planetside.co.uk/galleries/tg-in-film
[Accessed 13 10 2015].

Qt, 2016. *Qt Based Games.* [Online]
Available at: https://wiki.qt.io/Qt_Based_Games
[Accessed 20 01 2016].

Real-Time Rendering, 2008. *Direct3D 11 Details Part II: Tessellation.* [Online]
Available at: http://www.realtimerendering.com/blog/direct3d-11-details-part-ii-tessellation/
[Accessed 20 01 2016].

SDL, 2016. *Simple Directmedia Layer.* [Online]
Available at: https://www.libsdl.org/index.php
[Accessed 04 05 2016].

Shankel, J., 2000. Fractal Terrain Generation - Particle Deposition. In: M. DeLoura, ed. *Game Programming Gems.* Rockland, Massachusetts: Jenifer Niles, pp. 508-512.

Stranger, K., 2006. *Algorithms for Generating Fractal Landscapes.* [Online]
Available at: http://staffweb.worc.ac.uk/DrC/Courses%202013-14/COMP3202/Tutor%20Inputs/Session5/Keith_Stanger_Fractal_Landscapes.pdf
[Accessed 14 10 2015].

Ulrich, T., 2000. Loose Octrees. In: M. DeLoura, ed. *Game Programming Gems.* Rockland, Massachusetts: Jenifer Niles, pp. 444-445.

Walton, J., 2014. *Return of the DirectX vs. OpenGL Debates (AnandTech).* [Online]
Available at: http://www.anandtech.com/show/7890/return-of-the-directx-vs-opengl-debates
[Accessed 13 10 2015].

Wikipedia, 2016. *DirectX.* [Online]
Available at: https://en.wikipedia.org/wiki/DirectX#DirectX_10
[Accessed 04 05 2016].

WINE HQ, n.d. *About Wine.* [Online]
Available at: https://www.winehq.org/about/
[Accessed 13 10 2015].

Wolff, D., 2011. *OpenGL 4.0 Shading Language Cookbook.* Birmingham: Packt Publishing.

Young, S., 2015. *What The Heck is a Fractal and How Does It Apply to Games?.* [Online]
Available at: http://www.escapistmagazine.com/articles/view/video-games/columns/experienced-points/13809-Here-is-How-Fractals-Apply-to-Procedurally-Generated-Games
[Accessed 13 10 2015].