**Project Final Report**

Real Time Fractal Landscape Flyover

Submitted for the BSc in *Computer Science with Games Development*

April 2011

by

***Matthew Harris***

# Table of Contents


1 Introduction ..... 4
1.1 Initial Brief ..... 4
1.2 Context ..... 4
1.3 Aim and Objectives ..... 4
2 Project Background ..... 6
2.1 Problem Context and Ethics ..... 6
2.1.1 Problem Context ..... 6
2.1.2 Ethics ..... 7
2.2 Comparison of Technologies ..... 7
2.2.1 Rendering Technology ..... 7
2.2.2 Programming Language ..... 7
2.3 Alternate Solutions ..... 8
2.3.1 Alternatives to Fractal Terrain Generation ..... 8
2.4 Comparison of Algorithms ..... 11
2.4.1 Creation of landscape geometry ..... 11
2.4.2 Terrain Rendering ..... 12
2.4.3 Tree/Foliage Rendering ..... 14
2.4.4 Texturing Methods ..... 15
3 Technical Development ..... 17
3.1 Early Prototype ..... 17
3.1.1 Voronoi Heightmap and Perlin Noise ..... 17
3.1.2 Voronoi Heightmap with Height dependent shading ..... 17
3.1.3 Early Fractal Algorithm ..... 18
3.1.4 Multiple Blocks ..... 18
3.1.5 Multiple Blocks with Texture Splatting ..... 19
3.1.6 Smoothed Edges and Culling ..... 20
3.1.7 Super Blocks added ..... 20
3.1.8 Feature Spots and Sky Box ..... 21
3.1.9 Airplane and View Distance ..... 21
3.1.10 Improved Terrain ..... 22
3.1.11 Completed Program ..... 22
3.2 System Design ..... 23
3.2.1 Classes Overview ..... 23
3.2.2 Block Generation Flow ..... 24
3.2.3 Threading diagram ..... 26
3.3 System Implementation ..... 26
3.3.1 Texture Splatting ..... 26
3.3.2 Lightmap ..... 28

3.3.3 Lightmap Continuity ..... 30
3.3.4 Fractal Terrain Generation with Diamond Square ..... 33
3.3.5 Infinite Scrolling ..... 35
3.3.6 Super Blocks ..... 36
3.3.7 Feature Spots ..... 37
3.3.8 Trees and Geometry Instancing ..... 37
3.3.9 TerrainBlockGroups and Batching ..... 38
3.3.10 Multithreading ..... 41
3.3.11 Linear Fog ..... 42
3.3.12 Water Rendering ..... 44
3.3.13 Airplane Simulation ..... 44
3.4 System Testing ..... 45
4 Critical Evaluation ..... 45
4.1 Project Achievements ..... 45
4.2 Further Development ..... 47
4.3 Personal Reflection ..... 47
5 Bibliography ..... 48
6 Appendices ..... 50
6.1 Appendix of Test System Specifications ..... 50
6.1.1 Test System A ..... 50
6.1.2 Test System B ..... 50
6.2 User Guide ..... 50
6.2.1 Installation ..... 50
6.2.2 Operation ..... 50
6.2.3 Shutdown ..... 51
6.3 Credits and Sources ..... 51
6.3.1 Software used ..... 51
6.3.2 Sources of Textures and Models ..... 51
6.3.3 Special Thanks ..... 51
6.4 Timeplans ..... 52
6.4.1 Initial ..... 52
6.4.2 Revised ..... 53

# 1 Introduction

The aim of this project is to implement a program that will create and display a never ending landscape of mountains and hills. In this report, a series of methods and algorithms will be presented. The purpose of combining these methods is to process and display a fractal generated landscape to the user through 3D rendering.
This document is a review of the entire project, including specification, research, prototypes, design, implementation and a final review. The report is aimed at computer science literates with knowledge of programming and applicable mathematics. Specific research topics will be explained later on in order to elucidate the reader.

## 1.1 Initial Brief

The project title and description, as provided unchanged from the original is:

Real-time Fractal Landscape Flyover

*Specification:*
*The aim is to create a simple simulation of an aircraft flying over a complex fractal visualization of landscapes. The movement of the plane will be very simple. The creation and visualization of the landscape will be produced using a fractal algorithm.*

*Issues to consider:*
*1. Creation and storage of the geometrical data of the fractal landscape.*
*2. Rendering the fractal landscape in real-time.*
*3. Effect of rules on the creation of the landscape.*

## 1.2 Context

This is a 3D rendering application, and is not being designed for any user in particular, neither is it being developed in a team. It will use certain methods and algorithms which have been developed by other people, and these will be appraised later on in this document.

## 1.3 Aim and Objectives

In order to fully understand what is meant to be created at the end of this project, the title will be examined in detail and determine what each word means.

*Real-time* – the application must render the finished solution at over 25FPS (Frames Per Second) average on the target system.

*Fractal* – the terrain must be generated by some form of fractal iterative or recursive algorithm. The input and output can be non-fractal, as long as the core generation functions are. This means mountain positions can be input into the algorithm, then use a fractal algorithm to create the base geometry, and then use some sort of post process filter to make the terrain more believable.

*Landscape* – This at bare minimum means some form of terrain, but ideally the terrain should be big enough to scroll across the screen and make it feel as though the user is travelling across the terrain instead of just viewing an area of terrain.

*Flyover* – The application will allow control of a plane flying over the terrain with which to view the terrain in an intuitive way.

The core aims of the project are:

- Render a 3D terrain
    - Must be realtime (above 25FPS)
    - Must scroll as to appear larger than what is currently onscreen
    - Base generation of terrain must incorporate a fractal algorithm
    - Must be textured, ideally with some form of shader
- Control a plane flying over the landscapes

The secondary aims of the project are:

- Additional Terrain Detail
    - Include Trees on the terrain
    - Include Lakes and/or Rivers on the terrain
- Make the landscape scroll infinitely (if possible)
- Have the application scale graphics settings based on hardware available

This project will be considered ‘complete’ if the core aims are achieved, but ideally the secondary aims should be completed as well, but these can be cut out if time restraints become too tight.

# 2 Project Background

The application is a 3D visualization simulation of a landscape, and there are several elements that are to be considered when programming this solution.

## 2.1 Problem Context and Ethics

### 2.1.1 Problem Context

This project concerns itself with rendering mountains and hills using fractal algorithms. The fractal algorithms will be presented later on in the document. The other aspect that is non-computer science related is the nature of mountains. Some reference pictures will be used to model features of the landscape on.

**Figure 2-1 (Foxon C, 2010)**

This scene shows a snow topped mountain; note the jagged ridgeline at the top, and sheer faces. Snow is interspersed with rock, as opposed to being a flat covering.

**Figure 2-2 (Foxon C, 2010)**

Lower hilly areas are smoother, and often covered with trees, the overall shape of the hills are similar to mountains, but less jagged, and less steep.

These features will be taken into account when designing procedures to generate the landscapes.

### 2.1.2 Ethics

This project is a 3D rendering work, and does not include any third parties in collection of information. All resources used in creation of the report documents and application are referenced to their original creators, and any source code not created by the author of this document is labelled in the code itself. Any textures used are either purposefully created for this project, or from credited royalty free sources.

## 2.2 Comparison of Technologies

### 2.2.1 Rendering Technology

Mentioned previously in the document, it was said that DirectX would be used for the rendering portion of this coursework. The comparable technology that could be used as an alternative is OpenGL. The following is a comparison and justification of technology.

DirectX is a closed source API which is developed by Microsoft. It is usually the first to implement new technologies, because Microsoft works closely with hardware manufacturers. This makes DirectX the de facto standard for a large number of development studios, particularly any which do work on Xbox 360 (which uses a variation on DirectX 9.0). One of the big benefits to DirectX is that it provides easy interfaces to complex technologies, such as .fx files to describe combined vertex and pixel shaders.

OpenGL is an open source API which has a featureset managed by Khronos Group. The main benefit to OpenGL is that it is platform independent, so can run on Windows, Mac and Linux, as well as flavours on other platforms. It tends to be a little behind DirectX on new features, and cannot be used on Xbox platforms. There is a large amount of tutorial information on the internet for OpenGL, but a lot of it isn't relevant due to the new shader pipeline technology. There is more easily accessible documentation on DirectX through MSDN, but is less intelligible than many tutorial websites for DirectX.

The DirectX SDK will be used because it is simpler to use shaders with, and is a technology which is very widely used in the games industry, so familiarity with it is beneficial.

### 2.2.2 Programming Language

The two core languages with which the author feels comfortable programming in are C++ and C#.

C++ is an earlier language than C#, which is closer to the assembly code, which means faster performance than C#. This is paid for in code complexity and harder to debug code. However, the author has been programming in C++ for a few years more than C#.

C# is a more modern language that is conceptually easier to code in than C++. The disadvantage to this is that C# code usually runs slower than C++ code, particularly for CPU bound applications such as matrix multiplication (Sestoft, 2010). The XNA framework would allow code to run on PC and Xbox 360, but that isn't a requirement of the initial brief.

C++ is the language that will be used in this project because of the author's familiarity with the language, and its speed advantage in a performance critical project.

## 2.3 Alternate Solutions

### 2.3.1 Alternatives to Fractal Terrain Generation

Although the nature of this project is Fractal Terrain Generation, there are alternate methods that can be used to generate terrain heightmaps. A couple of them are discussed below

#### *2.3.1.1 Fault Formation*

This algorithm generates a heightmap by repeatedly dividing the map in half, along a random line, and then altering the height differently on either side of this line. The first iteration would give two flat surfaces of different heights. By repeating this again and again, after several iterations, a noisy terrain heightmap starts to appear.

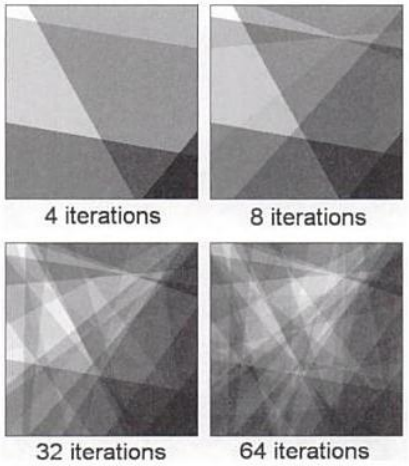



**Figure 2-3 Fault Formation Algorithm iterations (Shankel J, 2000)**

In order to end up with a reasonably realistic terrain, the amount each line offsets by is reduced with every iteration, so the first few iterations define the overall shape, and the later iterations add detail. After fault formation is complete, the map is often blurred to give a smoother, more realistic finish to the map. This algorithm is quite simple to implement, but isn't very versatile. It's difficult to control the output, and also needs lots of iterations plus blurring to make the final landscape acceptable.

#### *2.3.1.2 Particle Deposition*

Another alternative is particle deposition, which is a process by which a particle is deposited on the terrain, and falls to the lowest point of the terrain. Another particle is then dropped, and if it lands on top of another particle, it randomly goes in one of 4 directions, and keeps falling until it settles in a flat area.

**Figure 2-4 Particle Deposition**

As in the diagram above, the dotted particle hits another particle and moves to the left, as it's lower. The next particle hits the same place, and moves to the right this time, as the left is already flat. The next particle would just sit in place, as the left and right are both flat.
This algorithm is conceptually quite simple, but it often results in homogenous, almost spherical terrain, which doesn't look realistic.

### 2.3.1.3 Brownian Motion

Brownian motion (or Brownian Noise) is a function, which can be described using trigonometry which, with certain parameters, can be used to simulate the outline of terrain. The function can be defined as such:

$$B(t) = \sum_{f=-\infty}^{\infty} A_f r^{fH} \sin(2\pi r^{-f} t + \Theta_f),$$

**Equation 1 Brownian motion equation**

This results in the following, when given different parameters for H:

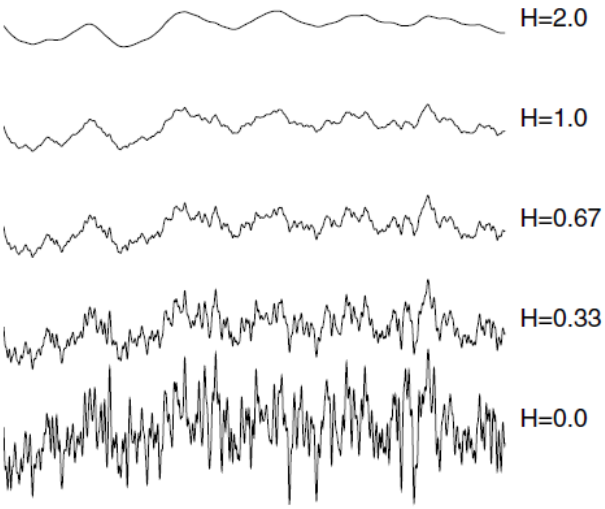



**Figure 2-5 Different parameter representations of Brownian Noise (Desussen O and Lintermann B, 2005)**

For values above 0.5, this function seems to appropriately reproduce the sort of contours associated with terrain.
This algorithm is good because it approximates terrain well, and can be evaluated at any point, so continuity is preserved. This would make it easy to have seamlessly connecting blocks. The problem with this algorithm is that it is somewhat complex to implement properly, and using a library function would negate credit. (Desussen O and Lintermann B, 2005)

### 2.3.1.4 Midpoint Displacement and Diamond Square

These algorithms need the 4 corners of a heightmap as input, and then recursively iterate over smaller and smaller squares of the map, randomizing values by a decreasing range of offsets.

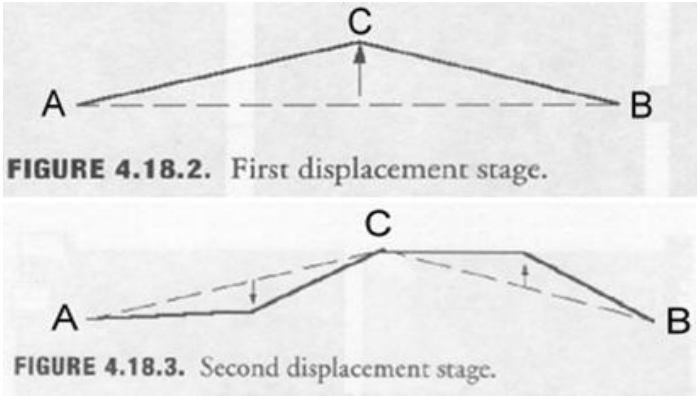



**Figure 2-6 Diagram of midpoint displacement (Shankel J, 2000)**

This method is good because it is very quick to execute, simple to understand, and the algorithm can be seeded by presetting heights in the map. The disadvantage is that the terrain generated can sometimes have square creases in it, damaging the realism effect. This is the algorithm that shall be used in the project, because it is fast enough to be done in realtime, and can be controlled, which means feature points can be set, and control where mountains/lakes appear.

#### *2.3.1.5 Voronoi Diagrams*

Voronoi diagrams are a method of dividing up a space by creating 'cells' around sets of points in that space. For example, in a 2D Voronoi Diagram, the space can be thought of as an image, with a series of random dots placed on its surface. Then, by creating bisection lines between each of those points, a diagram is formed where all pixels which are nearest to a particular dot are part of the same cell. An example diagram is below.

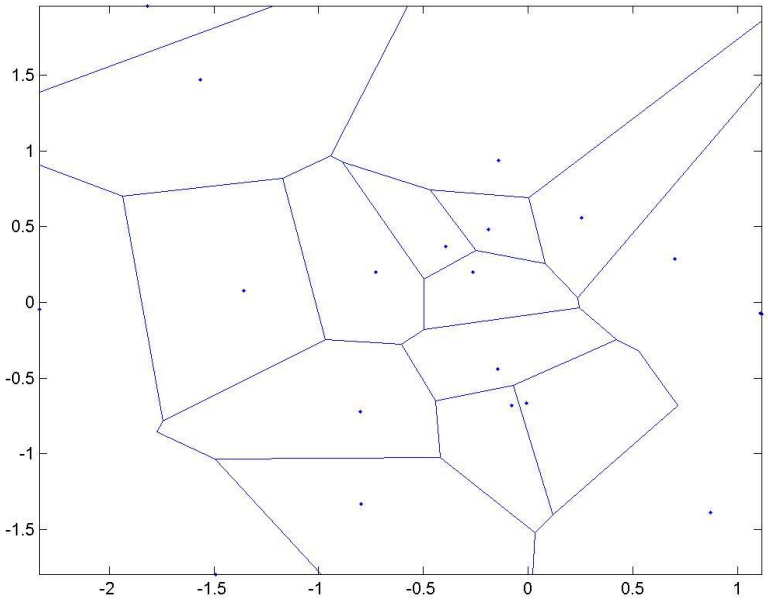



**Figure 2-7 Example Voronoi Diagram (Burkardt J, 2001)**

This diagram has divided up the image into separate cells, but this doesn't represent a heightmap yet. If an algorithm takes each of these cells and fills a centralised gradient in each, then every cell becomes a mountain. This approach was suggested in Realtime Procedural Terrain Generation (Olsen J, 2004), from which the below diagram is taken.

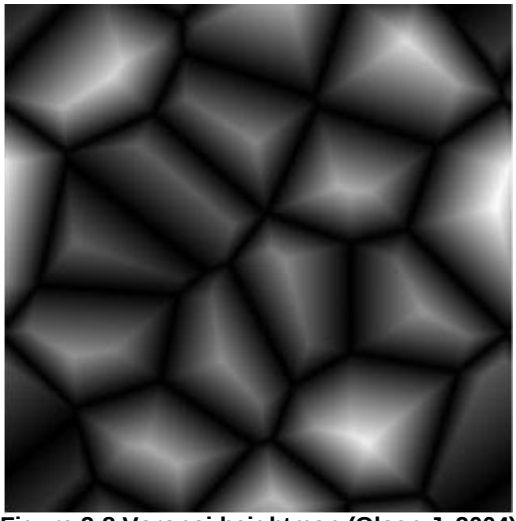


**Figure 2-8 Voronoi heightmap (Olsen J, 2004)**

Another improvement suggested in (Olsen J, 2004) is multiplying the max height of each by a random value, allowing mountains of different heights, with particularly low values becoming flat plains to allow gameplay on.

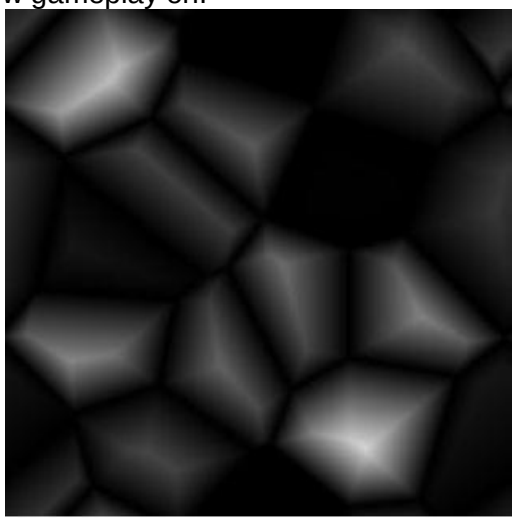


**Figure 2-9 Scaled Voronoi heightmap (Olsen J, 2004)**

## 2.4 Comparison of Algorithms

### 2.4.1 Creation of landscape geometry

As described in the project description, the creation of the geometry should involve a fractal algorithm. A fractal algorithm is a recursive or iterative function which repeats the same set of commands over a data set to get a pattern which is self similar. In that the overall pattern is mathematically similar to a subset of that pattern.
There are a number of different fractal algorithms which have been used to generate terrain geometry, some of which will be discussed below.
One of the most popular algorithms for fractal generation of a heightmap are the 'displacement algorithms'. These work by iteratively displacing the nodes in a heightmap by a decreasing amount, with the early iterations defining the overall shape of the mountain and the later iterations adding detail.
Research done so far has discovered two ways of doing this, Midpoint displacement, or diamond-square. Both algorithms start off with an array of points, and then the values for the 4 corner values are set as seed values.

Midpoint displacement takes these corner values, and makes 4 midpoints based on the

averages of the corners. After finding this average, the height is offset by a random number, creating a contour in the heightmap. This process is repeated, each iteration taking the previous square of 4 corners, and splitting it into 4 smaller squares, and performing the algorithm again. Each iteration reduces the range of random numbers that the point can be displaced by, making the initial points describe the overall shape, and the later generated points fill in the details.
The main problem with this algorithm is that it tends to make creases in a square shape due to the way the algorithm breaks down the array into squares.

Diamond square algorithm is an alternative to Midpoint displacement that is very similar, but slightly refined. There are two steps of iteration here. The first step is the 'diamond step' where the 4 corners of the square are averaged together to find 1 centre value, which is then randomly offset. Then the 'square step' is performed, where the 4 side points are calculated from the new centre point and the existing corner points. This proves to make more appealing terrain than Midpoint Displacement because the extra step calculates the new value on a diagonal, making the crease effect less noticeable. The disadvantage with this algorithm is that you sometimes get 'spikes' where the centre value is offset by a large amount and the terrain looks unrealistic. This should be avoidable by careful parameter selection, but research will continue during development for further refinements of this algorithm.
(Martz P, 1997)

Both algorithms will produce a heightmap following a similar pattern to this:

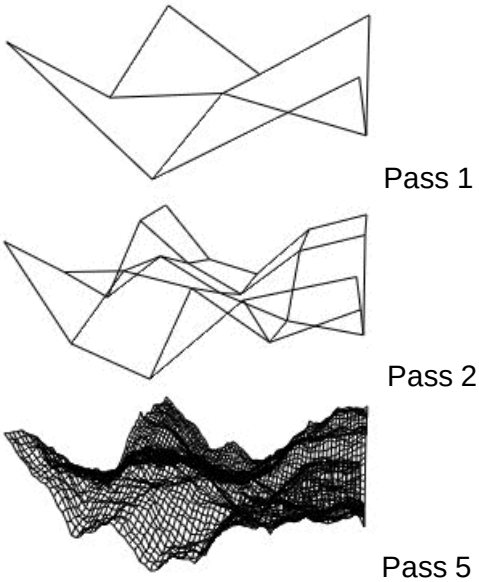



**Figure 2-10 Midpoint displacement over several passes (Martz P, 1997)**

The final pass 5 still has the same overall shape of pass one (note the leftmost corner is still high up), but the further passes of the algorithm have added detail to the mesh.
Diamond Square will be used instead of simple Midpoint Displacement, as it eases some of the square crease artefacts.

### 2.4.2 Terrain Rendering

Once the heightmap is generated, it needs to be rendered to the screen. The simplest way of doing this is do take every point on the heightmap, assign quads to link it to the other points around it, and send it all to the GPU. This is very expensive because a lot of polygons are used, and there are a lot of parts of a scene that don't need full detail (for example, behind the camera, where no terrain will be on screen). In order to make rendering fast, an

algorithm to simplify the mesh must be used, along with culling algorithms to cut down on superfluous triangles.

#### 2.4.2.1 ROAM

Realtime Optimally Adaptive Meshes is an algorithm developed in the 1990's as a combined CPU and GPU method for rendering terrain. The principle used is that the CPU scans over the heightmap, adding detail where it's needed (on bumpy contours) and adding only small amounts of detail where it isn't (on flat plains). This results in an optimal mesh which the GPU can draw, making the GPU render less triangles makes it render faster. The problem with this algorithm is that because it was developed a long time ago, it's not particularly efficient for today's modern GPUs. The reason for this is that every single frame, the mesh, or parts of it, need to be sent to the GPU again. Today's GPUs are much faster if they're rendering something in video memory, and updating that video memory every frame can be quite slow.
(Duchineau M et al 1997)

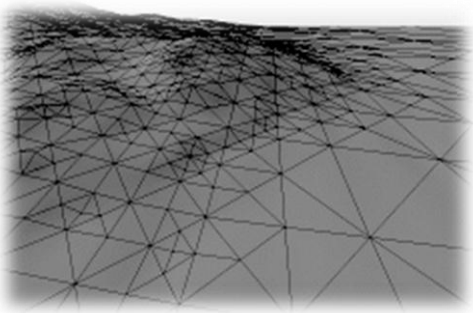


**Figure 2-11 Rendering of the variable detail tessellation ROAM provides (Duchineau M et al 1997)**

#### 2.4.2.2 Geometry Clipmaps

Geometry Clipmaps are a GPU-heavy solution which uses grids of different levels of detail which move with the camera over the terrain. As a piece of the terrain gets nearer, it becomes more detailed as it moves into the closer toroid grids. It uses modern GPUs efficiently, and also supports terrain compression and automatic geomorphing (so there's no level of detail pop in).

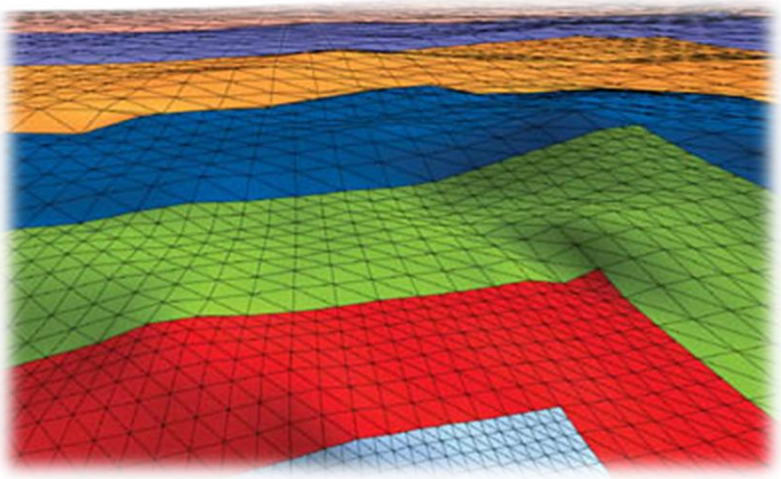


**Figure 2-12 Visualization of concentric toroids showing the decreasing terrain complexity as distance from camera increases (Asirvatham A and Hoppe H, 2005)**

The main problem with this algorithm is implementation complexity. It requires a lot of shader

code that may take the author a long time to understand. *(*Asirvatham and Hoppe, 2005)

#### *2.4.2.3 Geo Mip Mapping*

This algorithm is quite simple for implementation, and consists of breaking down the terrain to be drawn into patches (say, 64x64 points) and calculating different levels of detail for each patch (say 32x32 and 16x16 detail). When the patch is far away from the camera, a low detail patch is displayed, and when neared, a high detail patch is swapped in. This is a simple algorithm to implement, but has a couple of caveats. One is lining up different patches to prevent gaps or T-junctions. The other is 'pop in'. Pop in is when a patch is swapped to a different detail instantaneously, and the player sees the extra polygons jump into view. (de Boer W, 2000)

In this project, Geo Mip Mapping will be used, due to its simplicity and that it should work well with a system of block-based terrain generation. Geo Clip Maps are too difficult to implement fully on this time scale, and the performance gains are unknown.

### 2.4.3 Tree/Foliage Rendering

#### *2.4.3.1 Rendering Grass as Textured Grass Clusters*

This is a method by which grass objects are constructed with a semi transparent texture, showing several blades of grass is used. Each of these objects consists of only a few faces, meaning that there are very few polygons to render per object. This means that lots of them can fit in the scene.

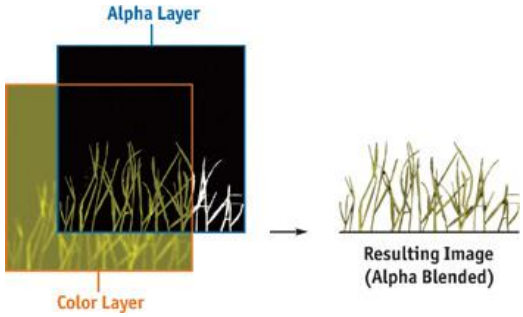



**Figure 2-13 An example of a grass texture that allows multiple blades of grass per face (Pelzer K, 2004)**

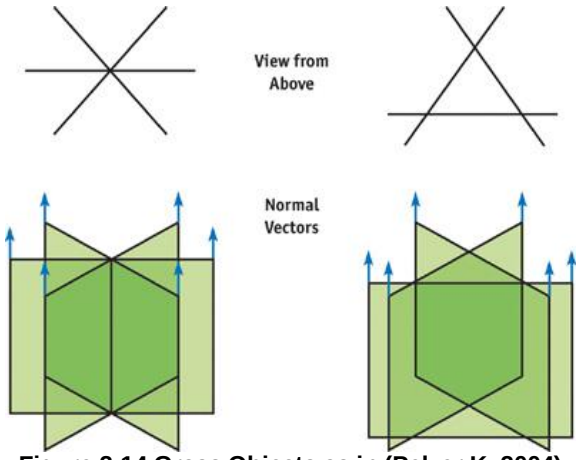



**Figure 2-14 Grass Objects as in (Pelzer K, 2004)**

Using vertex shaders, the grass objects can be animated separately using basic trigonometry functions. If each object knows the central point of the object, then the top vertexes can be moved in unison, so as not to distort the texture. (Pelzer K, 2004)

#### *2.4.3.2 Geometry Instancing*

In order to cover a terrain in tree objects, a lot of them need to be drawn, in the order of hundreds on screen at the same time. If each of these were to take up an individual draw call, then the CPU would grind to a halt. In order to render large numbers of identical objects quickly, instancing can be used. A packet of objects is assembled, and that whole set of objects is rendered at once with one draw call.

**Figure 2-15 Instancing used in Black and White 2, referenced in GPU Gems 2 (Carucci F, 2005)**

As seen above, instancing allows hundreds of identical objects to be rendered quickly. The above image shows people, however using trees or cacti models instead would lead to a convincing looking forest or foliage group. (Carucci F, 2005) (Gosselin D et al, 2005)

Geometry Instancing will be the first technology used to render foliage in the project, due to its quick setup time and versatile nature (can be used for trees or grass clumps). However, waving blades of grass may be implemented if time permits.

### 2.4.4 Texturing Methods

#### *2.4.4.1 Texture Splatting*

This technique is a method of using a low res texture map called a splat map which is used to alpha blend several other textures in a smooth way. This means that instead of using one large texture (eg 512x512), a small splatmap (32x32) and a couple of small textures (64x64) repeated over the geometry, which greatly reduces bandwidth consumption, but allows lots of extra detail in the scene.

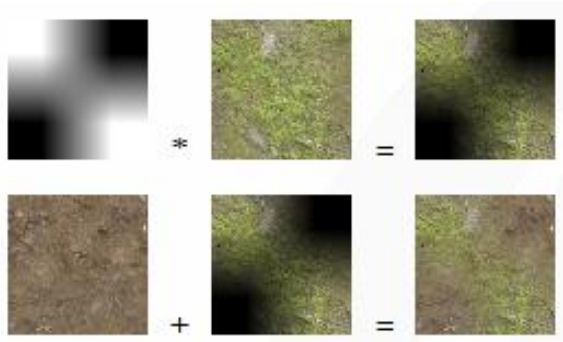


**Figure 2-16 (Glasser N, 2005)**

As shown above, the greyscale image is the splatmap, which is only 4x4 pixels wide, and using it to multiply by the texture, the third image is obtained, of grass with black faded gaps. When combined with the dirt under-layer, the final image is obtained, which is of smoothly transitioned grass and dirt.
This technique was originally described by (Bloom C, 2000) and pixel shader principles described by (Glasser N, 2005)

# 3 Technical Development

This section will cover the process of creating the final application, beginning with early prototypes and finishing with the completed simulation.

## 3.1 Early Prototype

### 3.1.1 Voronoi Heightmap and Perlin Noise

As early research was ongoing, some experimental heightmap generation techniques were investigated. One research paper (Olsen J, 2004) suggested that Voronoi diagrams could be used to create mountainous sections. In the below prototype, the terrain is split up into a number of voronoi areas of differing heights, and then multiplied by a perlin noise factor. The multiplication means that high areas appear rough and rocky, whereas lower heights are flatter and smoothed, with the noise adding only fine detail.
The top left shows the normal map of the surface.

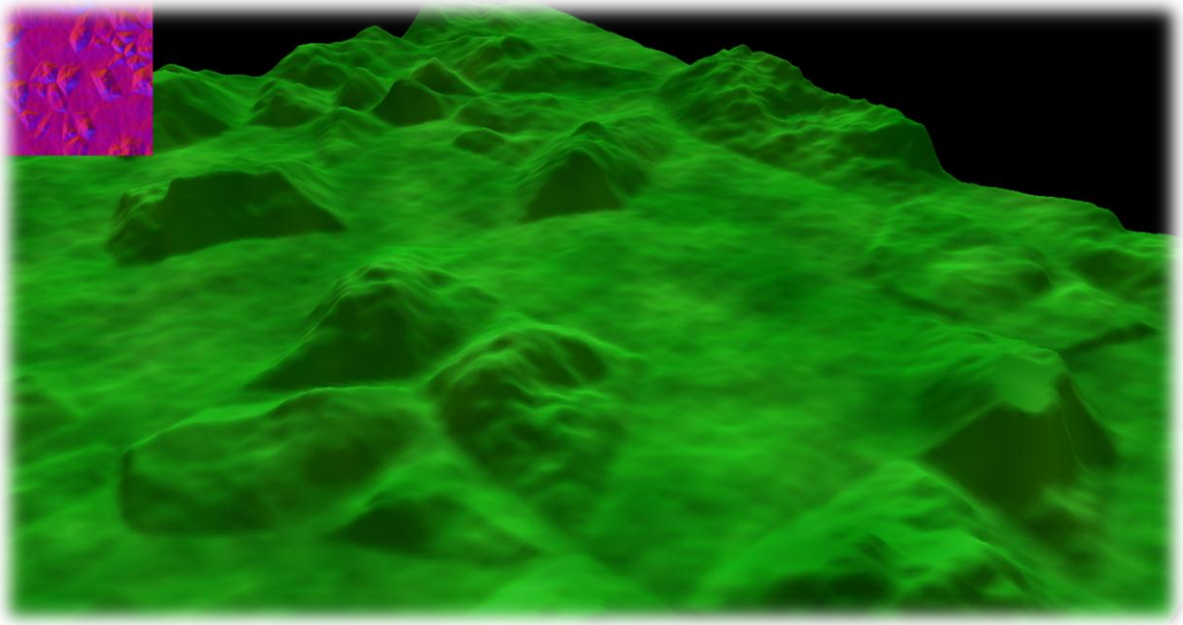


**Figure 3-1 Screenshot of Prototype 1**

### 3.1.2 Voronoi Heightmap with Height dependent shading

This was an evolution of the prototype above, with some HLSL texturing applied. The method used was linear interpolation of different texture samplers based on height, so that the higher a vertex was drawn, the less of the grass texture was used, and more of the snow texture was used.
Also visible in this screenshot is the base grass texture, which is just a green noise texture, similar to Perlin Noise. It was initially planned that the final product might also generate all textures procedurally.

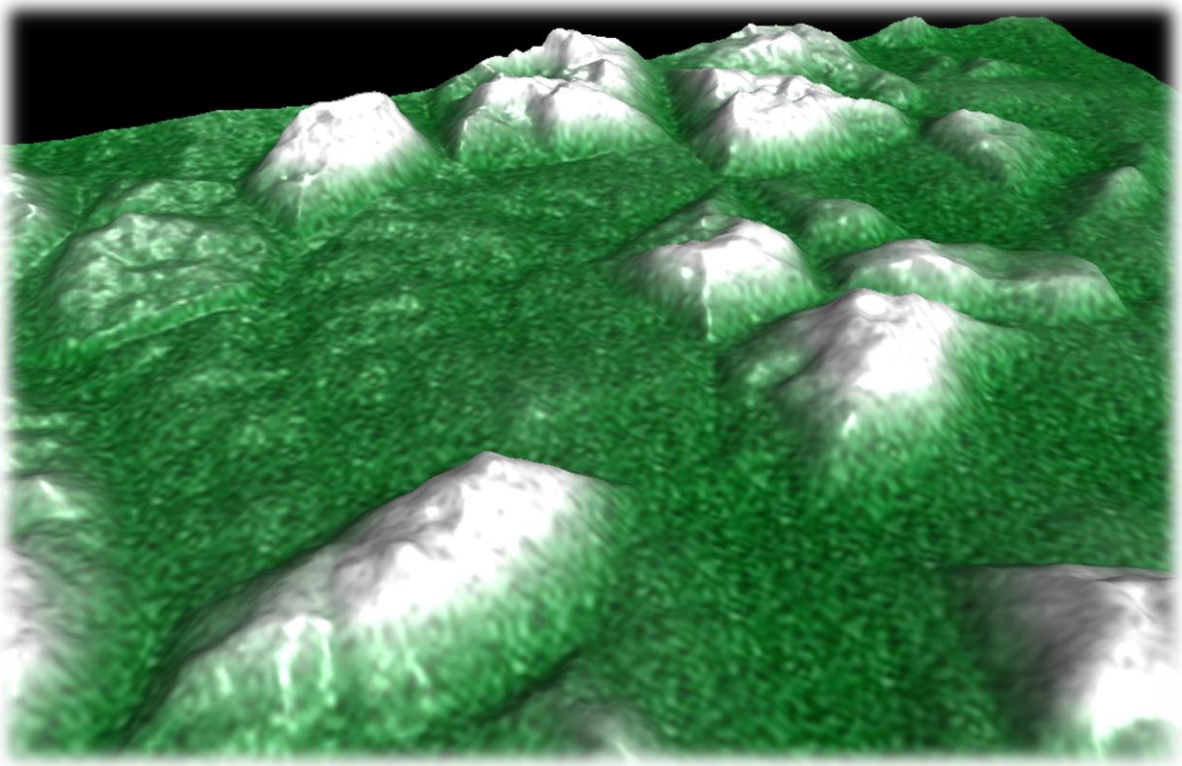


**Figure 3-2 Screenshot of Prototype 2**

### 3.1.3 Early Fractal Algorithm

Although the voronoi approach produced decent results, the flats were too boring, and the mountains too sudden. In the below prototype, a diamond-square fractal algorithm was employed. This allows for much greater detail to be generated, as the algorithm can be seeded with a few points, and extrapolate an interesting, ridged mountain shape between those points.

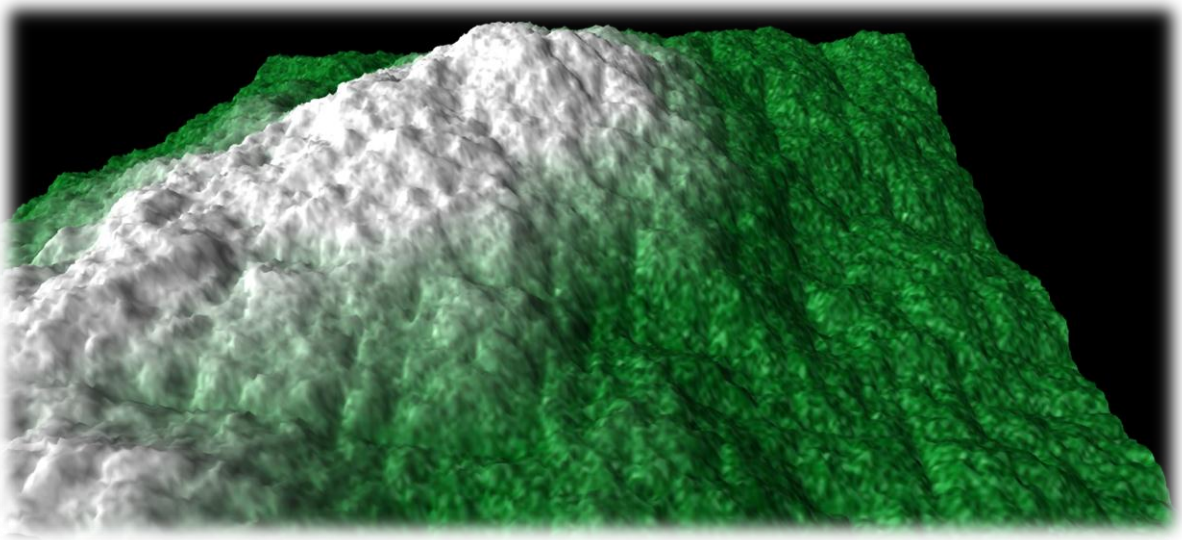


**Figure 3-3 Screenshot of Prototype 3**

### 3.1.4 Multiple Blocks

The final goal of the application is to have an infinitely scrolling world, and to do this, multiple blocks must be stitched together. Here, when a block is generated, it checks if any neighbouring blocks have been generated; if so, it uses that edge as seeds for the fractal algorithm, so that the edges match up. An early implementation of LOD is also shown here,

as the central block is at very high res, at 256x256 heightmap resolution, and the neighbour blocks are 8x8. Also shown is a graphical bug, where the normal for the lower resolution blocks are not filtered correctly, giving this striping effect, where each triangle strip can be seen. This was later corrected by averaging values from the high res map, and removing erroneous offsets from the code logic.

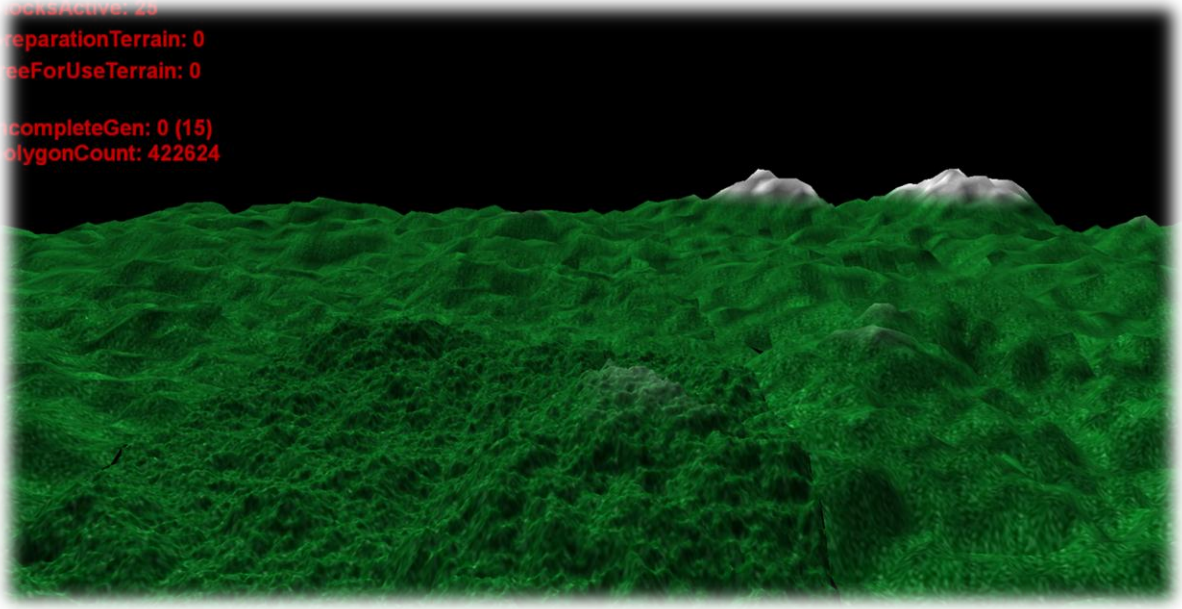



**Figure 3-4 Screenshot of application in development**

### 3.1.5 Multiple Blocks with Texture Splatting

Several new features have been added in this version. Firstly, texture splatting has been added, allowing 4 separate textures to be displayed on the terrain. In this shot, the sand smoothly transitions into grass and then into snow on the highest spots. Also implemented here is infinite scrolling, where the offset X and Y values are used to seed the fractal generation, and the blocks are dynamically scrolled around the origin, giving the illusion of smooth movement. The scale of this scrolling is only limited by the integer precision used to store the block X and Y coordinates, which makes the total number of possible blocks 4 billion squared. Note that the array of 'A's in the screen describes which of the blocks are active and have been generated fully.

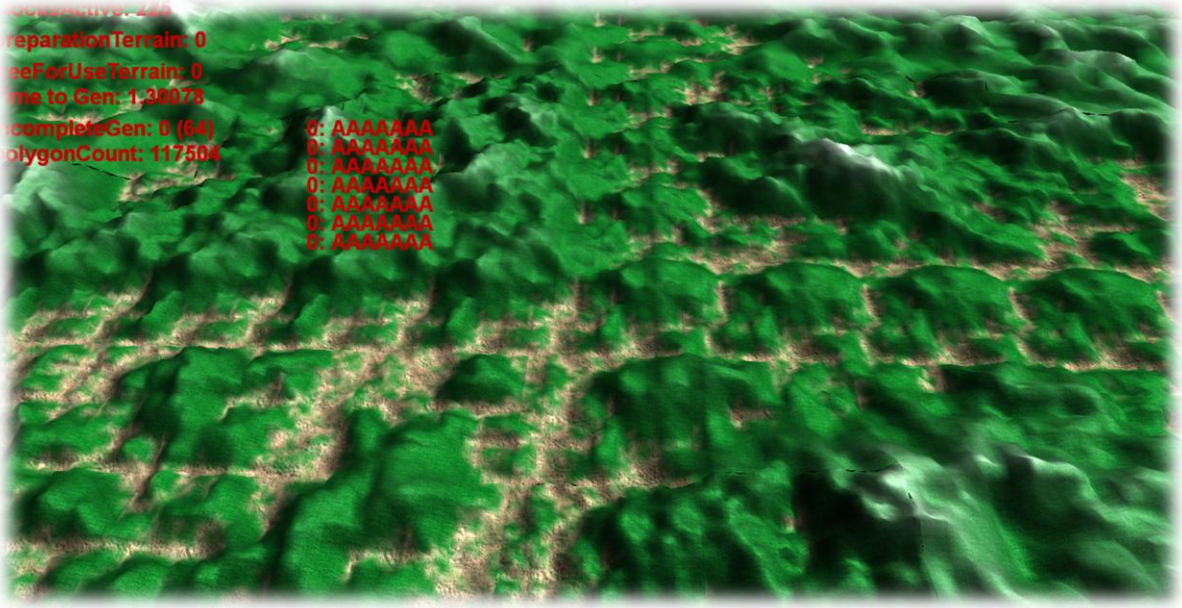


**Figure 3-5 Screenshot of application in development**

### 3.1.6 Smoothed Edges and Culling

Similar to the previous build, but now a culling frustum has been added, which vastly improves rendering performance. The dot matrix display on the screenshot shows blue when a block is inactive, and red when it's being displayed. There is a large amount of rendering draw calls saved here, which vastly improves the framerate.

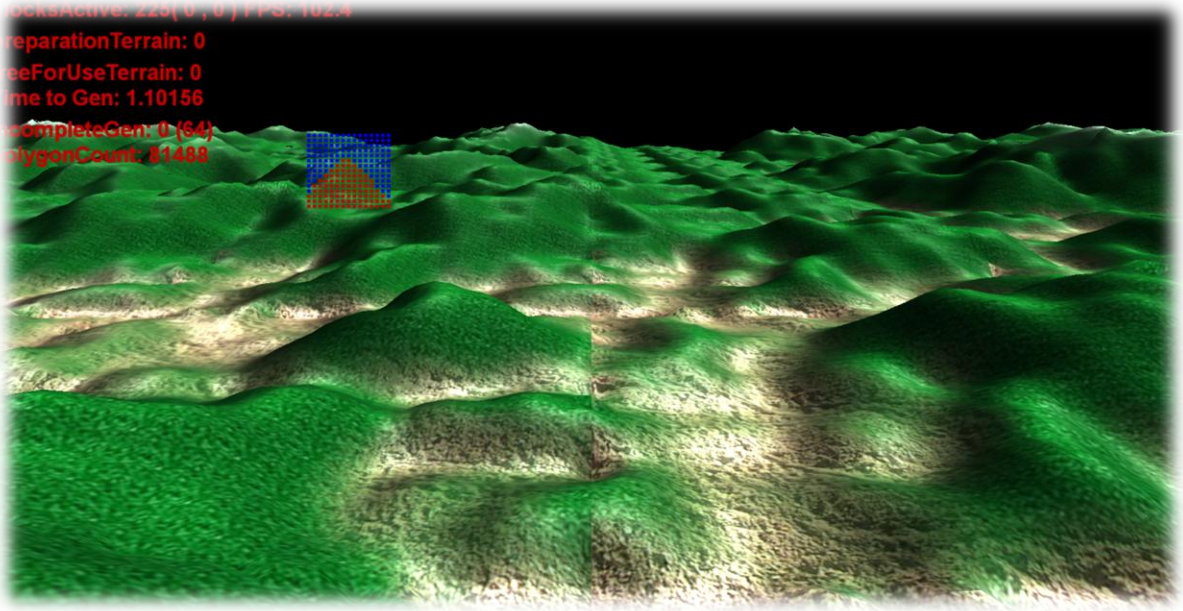



**Figure 3-6 Screenshot of application in development**

### 3.1.7 Super Blocks added

The main problem with the look of the application so far has been that there are no overarching features such as mountains or valleys, just homogenous lumpy hills. In order to correct this, Super Blocks are added. These are 128x128 sized heightmaps that are also fractal generated. These heights are then used to seed 64x64 blocks (scaled down to allow the superblock to describe the central 'diamond step' of the block. This means that patterns that are larger than a single block can be generated, while retaining the fractal nature of the generation algorithms.

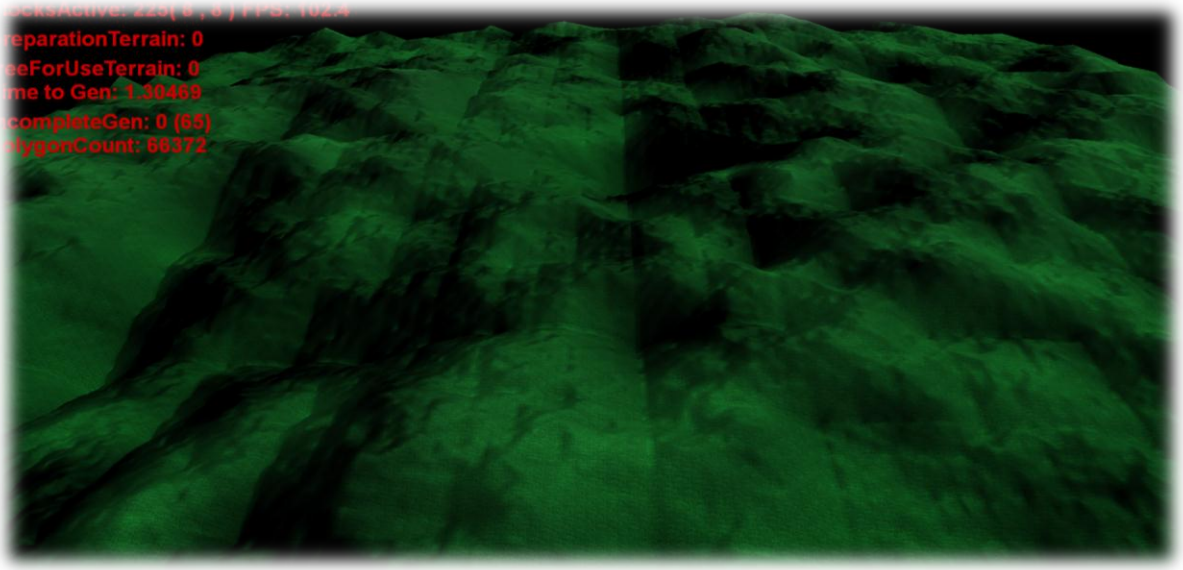



**Figure 3-7 Screenshot of application in development**

### 3.1.8 Feature Spots and Sky Box

In this build, the Super Block code has been modified to produce feature spots. These spots are randomly selected in size and position, and could be a bowl shaped lake, flat area for a forest or a volcano shape (as pictured). The Super Block now also has the ability to tell underlying blocks what texture splat bias they should use. This allows volcanoes to use a new rock texture, which is also pictured.

A sky box has also been added, which makes the terrain feel as though it is part of an actual world, rather than floating in a black void. This was achieved with a non-depth tested cube that moves with the camera to give the illusion of infinite distance.

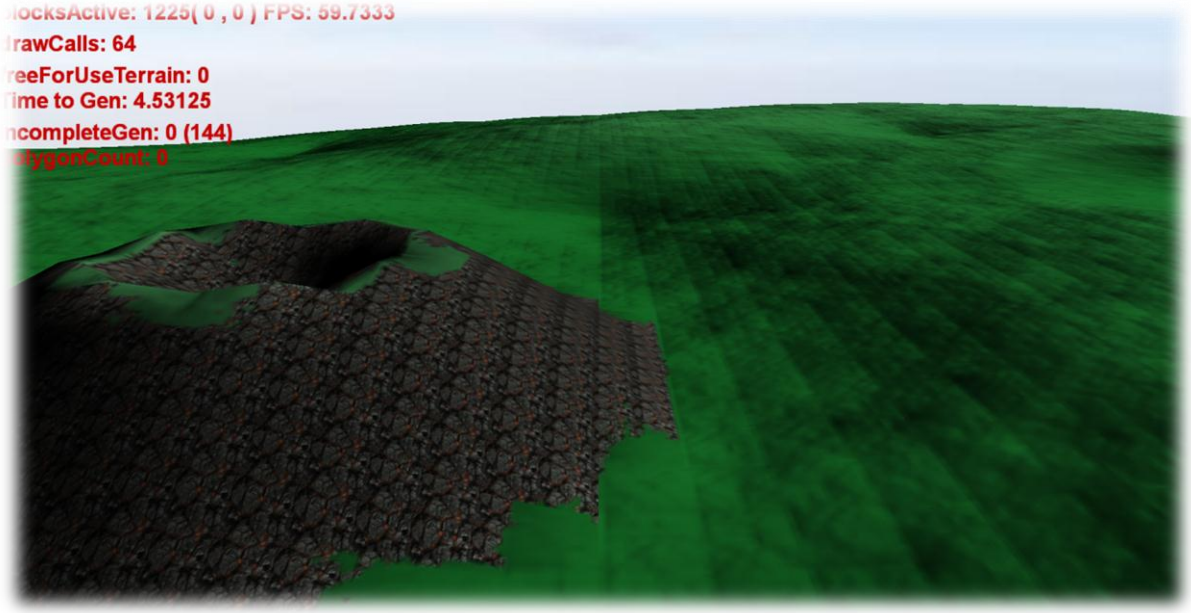



**Figure 3-8 Screenshot of application in development**

### 3.1.9 Airplane and View Distance

A simple model airplane has been added and allows smooth and intuitive navigation of the terrain. The Super Block code has also been modified to provide smoother transitions between textures when creating volcanoes.

In addition, when detailed geometry hasn't been generated yet, the underlying superblock is used to show a low detail version of the terrain further away to maintain the illusion of a solid surface.


**Figure 3-9 Screenshot of application in development**

### 3.1.10 Improved Terrain

The Super Block generation code has been modified to give much greater peaks and troughs, allowing realistic mountains and valleys to be generated. This has instantly transformed how interesting the terrain is and improved the visual quality enormously. Also of note here is how the volcano has a slightly frosty peak, because it is generated high up near the snowy mountains.

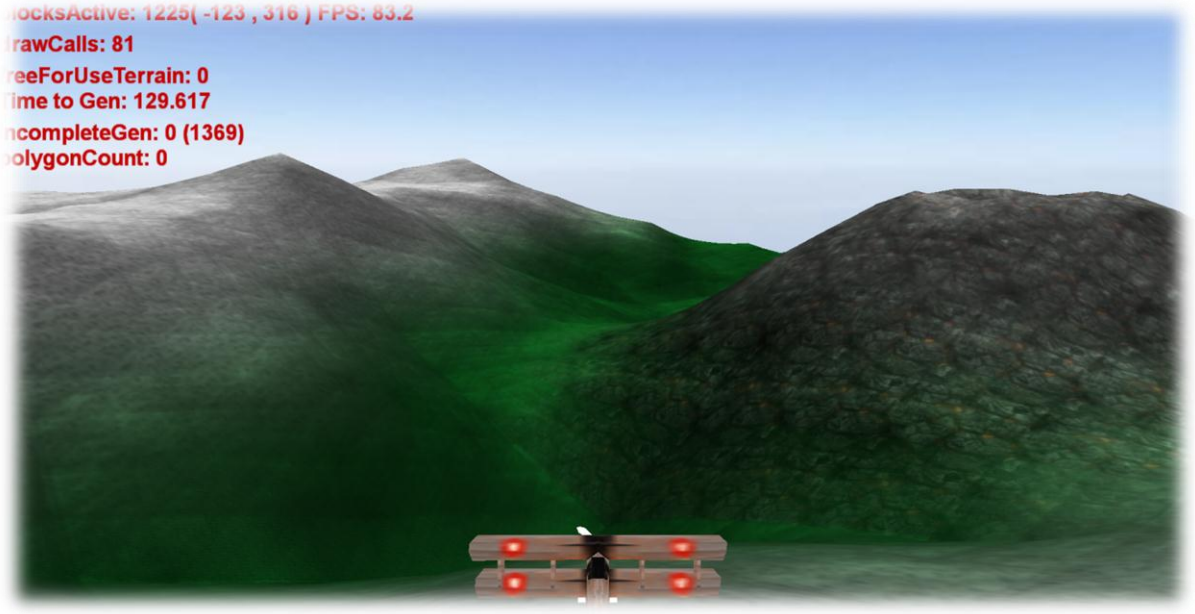



**Figure 3-10 Screenshot of application in development**

### 3.1.11 Completed Program

The final program has been improved with level of detail on TerrainBlockGroups, meaning that nearby blocks have more detail (evidenced by the rocks around the volcano). Linear fog has been added to disguise the view distance pop-in.

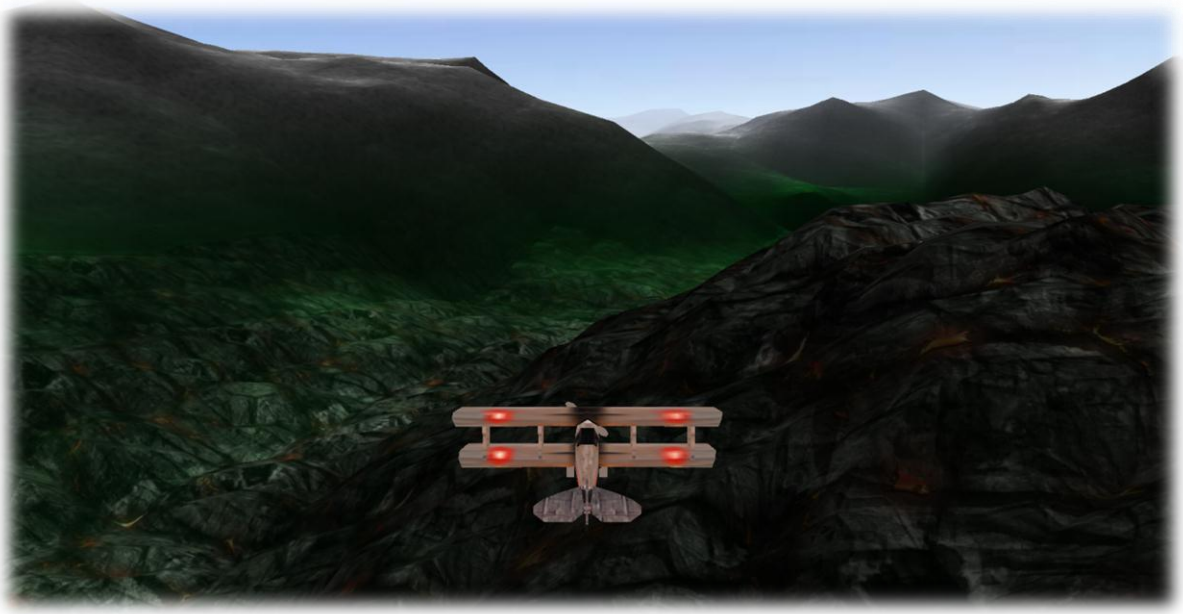


**Figure 3-11 Completed application**

## 3.2 System Design

### 3.2.1 Classes Overview

There are several separate classes involved in rendering this system, many of which are linked, although not by inheritance. The following class diagram presents a simplified overview of the classes in use in the final version of the system. The TerrainDX9 class isn't included as it's merely the starting framework, and creates an instance of TerrainManager to start terrain generation and rendering. In order to maintain diagram simplicity, only a few key variables and functions are included for each class.

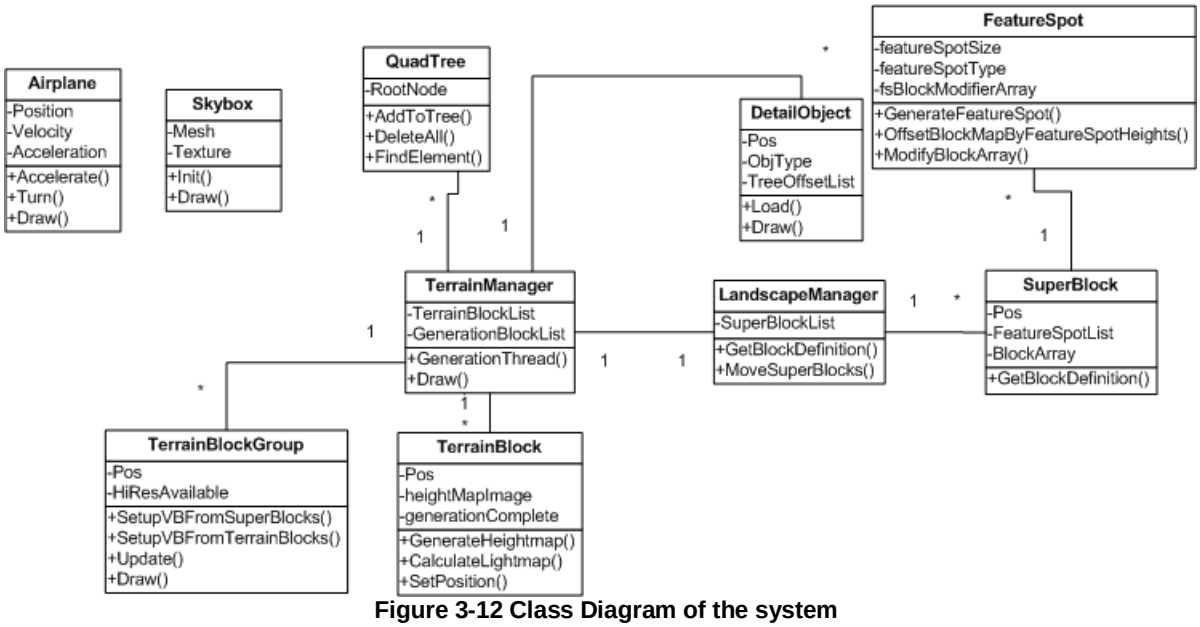



**Figure 3-12 Class Diagram of the system**

**TerrainDX9.cpp –** This is the main class file which sets up the window and directX initialisation. It coordinates the separate parts of the system, such as the airplane and skybox classes, and creates an instance of TerrainManager to start generation and display of the terrain.

**TerrainManager.cpp –** The heart of the terrain code. This class manages two arrays of TerrainBlocks, one for generation and one for display. Upon creation, this class spawns a separate thread, which runs in parallel with the draw and update thread. This new thread is the generation thread, and is constantly checking for when new blocks need to be generated. If so, it gathers data from the SuperBlock and previously generated TerrainBlocks to create the next block's heightmap.

**TerrainBlock.cpp** – This describes a 32x32 heightmap of terrain data, and stores the heightmap, splatmap and lightmap for this block. The splatmap describes the contributions of different textures to the final colour of the block. The lightmap is a texture component which is applied to the final render to give the terrain block some lighting. This class contains the fractal terrain generation algorithm along with functions to calculate the normal map, splatmap and lightmap.

**SuperBlock.cpp** – A larger scale description of what the terrain shape should turn out to be. This is like a blueprint for the TerrainBlocks to work from, and describes large scale features such as mountains and valleys. It uses its own fractal generation algorithm to make the mountain outlines, and stores this in a 128x128 grid of heights. It then randomly assigns FeatureSpots to itself and offsets the heightmap to accommodate these new areas of interest.

**FeatureSpot.cpp** – Describes interesting terrain deformation spots, such as a volcano, lake or forest. This class can be told to randomly pick a position and size within a heightmap (of a SuperBlock) and also modify the texture offsets. The offsets are simple equations, so a lake is simply a cone offset, so are not fractal. However, the FeatureSpot is able to label areas of the SuperBlock as having a modified fractal roughness for when the TerrainBlocks are generated. This means that volcanoes can have rough, rocky peaks, whereas lakebeds remain smooth.

**DetailObject.cpp** – A renderable object which describes supplementary details that are not part of the terrain heightmap itself. For example, a lake FeatureSpot makes a bowl shape in the terrain, and creates a DetailObject to store the lake water details. This class makes use of static variables to only load in the geometry for each type of object once. Each detail object just stores details about where an object should be placed, and what sort (e.g. positions of trees in a forest about a particular block X and Y).

**LandscapeManager.cpp** – Manages creation and deletion of SuperBlocks. When the camera moves so far that a new SuperBlock is needed, this class creates a new row of them, and removes old ones. There are only 9 SuperBlocks in memory, the currently displaying one, and the 8 around it.

**QuadTree.cpp** – A class which manages nodes of a QuadTree, and allows population and searching of a large amount of data in an efficient manner.

**SkyBox.cpp** – Loads the geometry for the skybox, and renders it at an appropriate position.

**Airplane.cpp** – Loads the geometry for the plane, and renders it at an appropriate position. It needs to receive calls to its Update method in order to update the plane's velocity and position. It also takes control commands such as Accelerate and Turn.

### 3.2.2 Block Generation Flow

In order to create an individual block, data from several other classes is required. A simplified flowchart outlining the process is as follows.

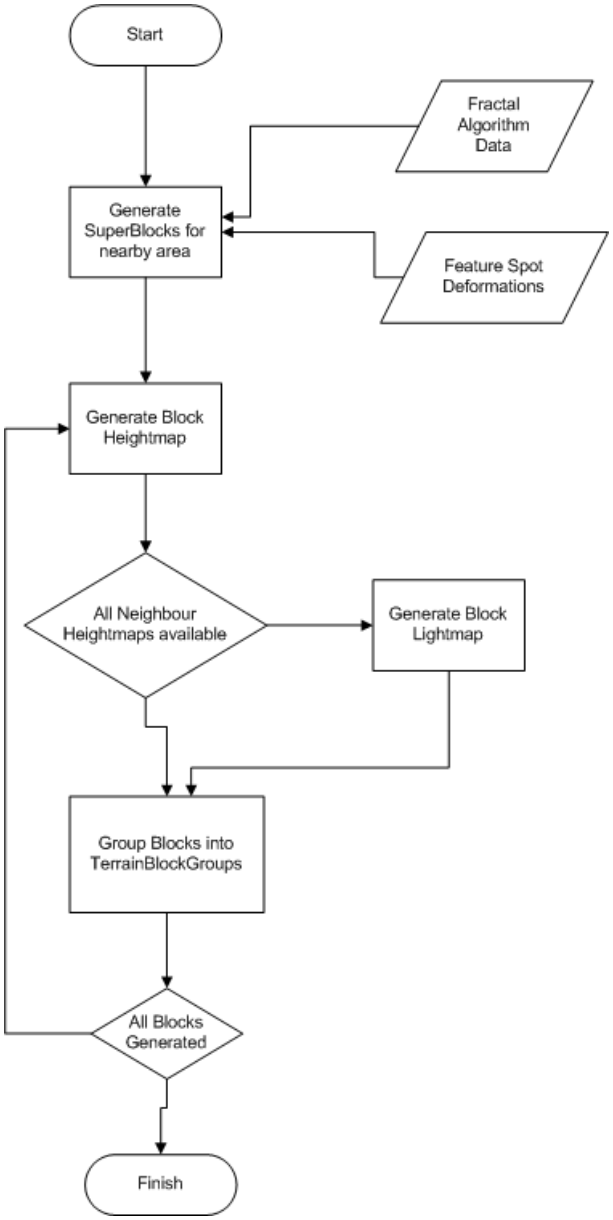



**Figure 3-13 Flow Diagram outlining block generation**

At the highest level, SuperBlocks need to be created to describe the outline of the terrain near the player. The basis of this is a fractal algorithm (Diamond Square), which describes the shape of the mountains and valleys. A set of feature spots are created, and then their effects are applied to the SuperBlock to offset the fractal geometry by the required amount.
Within TerrainManager's generation thread, it is periodically checking for valid blocks to generate. If it finds one, it takes the seed points from the superblock, along with any parameter offsets, such as increased roughness, and uses these variables to generate the heightmap for the block using the fractal diamond square algorithm. It then checks to see if creating this block has surrounded any blocks with other generated blocks. If so, the lightmap for this central block can be created. The lightmap can only be created once neighbours are present because the lightmap needs to take into account heights from outside its own heightmap in order to keep the edges of blocks smooth and non-apparent to the user.
After some blocks have created, the other thread checks to see if there are enough to be grouped up. A TerrainBlockGroup consists of combined geometry from 16 TerrainBlocks in a 4x4 square. When rendering, it is the TerrainBlockGroups that are drawn, rather than the TerrainBlocks. This is because there can be more than 1000 TerrainBlocks active at any one

time, and drawing each with an individual draw call will vastly reduce CPU performance due to DirectX overheads (Wloka M, 2003). The TerrainBlockGroups reduce this by an order of magnitude, allowing for much better efficiency.

### 3.2.3 Threading diagram

The approach taken to multithreading is discussed in greater detail in 3.3.10, but for system design purposes, a sequence diagram, showing an outline of the thread's interactions is produced below.

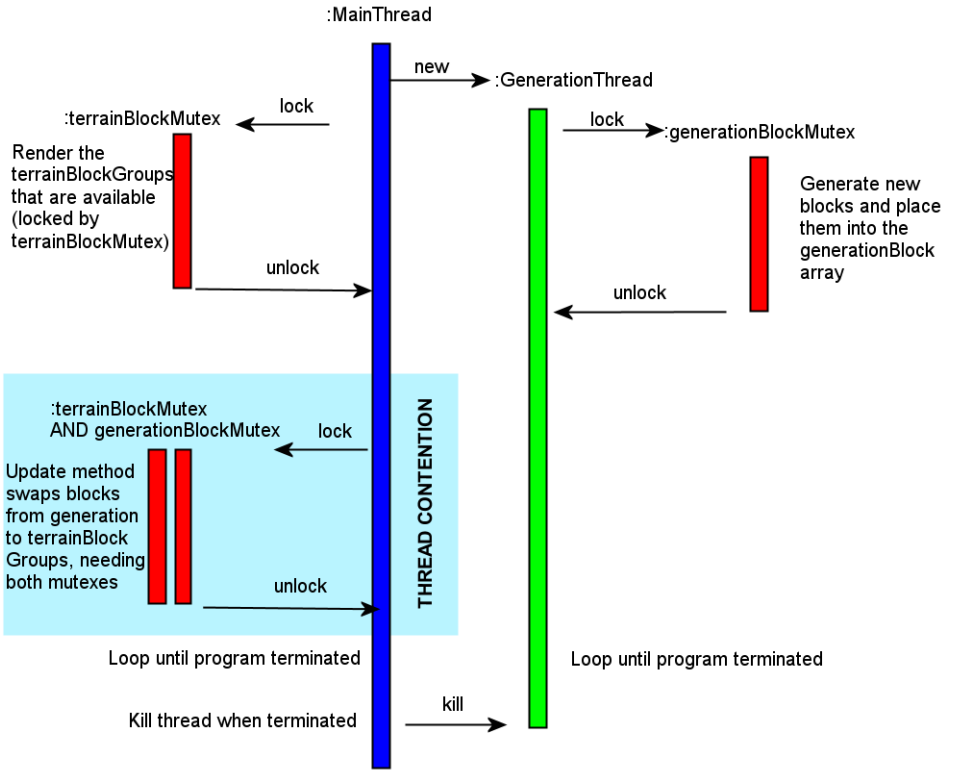



**Figure 3-14 Thread behaviour diagram**

The MainThread creates a new GenerationThread to take over geometry creation, while the original thread is responsible for drawing and updating the TerrainBlockGroups. This means that while the GenerationThread is busy generating blocks with one mutex locked, the MainThread can be drawing them to the screen with the other mutex locked. The key part of this process is where the MainThread needs to lock both mutexes in order to copy data from the generationBlocks into the TerrainBlockGroups. This is done regularly on a non-blocking mutex lock, in order to stop the MainThread from stalling. This ensures that rendering framerate remains smooth, but at the cost of large pop-in of detail. Both threads loop until the program is terminated, upon which the MainThread sends a kill command to the GenerationThread, which frees up any resources it has allocated and then terminates.

## 3.3 System Implementation

### 3.3.1 Texture Splatting

As described in 2.4.4, Texture Splatting is a technique where a low resolution texture is used to control which type of texture is used in the final colour of the object. It is often used in terrain rendering to allow smooth transition between grass and dirt or other ground textures.
In this project's implementation of texture splatting, it was aimed that 4 textures should be used: grass, sand, rock and snow. This way, the desert areas are sandy, the hills are grassy, mountain peaks are snowy and volcanoes are rocky.

**Figure 3-15 The four primary textures used in the project**

Each pixel in the splatmap texture consists of 4 values (channels): RGBA, which vary from 0 to 1. The obvious way to implement texture splatting would be to have each channel describe how much each texture contributes to the final image, so R=Rock G=Grass B=Sand A=Snow. So for an entirely grass splat, RGBA=0 1 0 0 and for a half grass half snow splat RGBA=0 0.5 0 0.5. However, this method ignores the situation of RGBA = 0 0 0 0, which would end up being black, which isn't a useful texture.
Instead, this implementation assumes that grass is the primary texture, and then uses RGB for R=Sand G=Rock B=Snow, with the A channel free to use for the lightmap, which will be discussed later. This means that each channel takes the current value (which starts off at grass) and linearly interpolates it towards the value of that texture sampler. So if RGB=0.5 0 0 is used, then the shader starts off with a grassy colour, and moves 50% towards sand, resulting in a sandy-grass effect. A side effect of this method is that later channels are dominant, so RGB= 1 0 1 takes grass as it's starting point, moves entirely to sand, and then entirely to snow, so the final texture is just snow. This can be useful to ensure caps are snowy, but care needs to be taken when setting the splatmap values to avoid visual artifacts.

```
float4 splatColor = tex2D(splatSamp, inStream.Texture);
float snowFactor = splatColor.b;
float rockFactor = splatColor.g;
float sandFactor = splatColor.r;

float4 texColor = float4(grassTexColor);
texColor = lerp(texColor,sandTexColor,(sandFactor));
texColor = lerp(texColor,rockTexColor,(rockFactor));
texColor = lerp(texColor,snowTexColor,(snowFactor));
```

As seen above, the code for this effect is very simple and efficient. The outline of the idea and linear interpolation concept is taken from (Glasser N, 2005).
The main problem with this method is its lack of versatility. Each splat can only be coloured according to 4 textures, so more complex effects aren't possible unless another texture is used to add another 4 channels to the splatmap palette, or each channel is split in 2, which would reduce precision. The alternative method to add variety is to have different texture sets for different TerrainBlocks, which means SuperBlocks can use different sets of 4 textures. An example of this is the desert area. Snow doesn't have any place in a desert, so the snow channel of the splat map would be unused. In desert areas, the desert texture set is used, so that the snow channel actually describes a second type of sand texture, allowing greater visual fidelity to be achieved.

**Figure 3-16 Desert area with two distinct sand textures blended together**

The caveat here is that if two completely different texture sets are used in blocks beside each other, there will be a clear straight line between the two blocks as sand suddenly turns into grass, or another combination. To solve this, splatmaps of blocks at the edges of superblocks are forced to fade to grass before meeting up with the neighbouring superblock, giving a fading grass effect, which is much more appealing.

**Figure 3-17 Transition from grass to desert over a SuperBlock boundary**

### 3.3.2 Lightmap

Using the splatmap above gives each block a unique texture pattern, but lighting would then be left to the normal to shade this texture pattern into a surface that appears to have depth to it. For most small-scale objects, this method is fine, as polygons are close together, and normals provide enough detail for good visual quality. With terrain mapping, the main problem is that aggressive geomipmapping is needed. Geomipmapping is where lower detail

versions of the terrain are used when they are far away from the camera, and higher detail ones used when close to the camera. For example, a 2x2 vertex buffer, giving a very flat outline of the TerrainBlock would be used far off in the distance, and a 32x32 vertex buffer could be used near to the camera. This speeds up rendering by reducing the amount of polygons rendered. The problem with this method is that when the buffers are swapped between detail levels, the normals suddenly change, giving a large visual change in lighting, as parts of the terrain get brighter and darker to accommodate the new calculations.
A good solution to this is lightmapping. This technique sends a single channel of texture to the shader which describes whether this should be light (1) or dark (0). The terrain splatting algorithm already uses RGB, so the lightmap component can occupy the A channel, without adding any more data to the structure to be sent across the GPU bus.
To calculate the lightmap, normal calculation must be performed on the heightmap, the simplest way to calculate a normal is as follows:

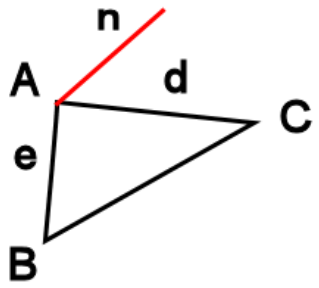



**Figure 3-18 Calculating a normal from three points**

If A,B,C are the points on the heightmap, forming a right-angled triangle, then **d** is the vector from A to C and **e** is the vector from A to B. To find **n**, the normal of the triangle, the cross product is used:

$$\boldsymbol{n} = \boldsymbol{d} \times \boldsymbol{e}$$

**Equation 2 Cross Product example**

The problem with this method is that it doesn't take into account the points nearby, and so linear patterns start appearing in the normal. A better approach is to take an average of the points around the centre vertex, meaning that more data is taken into account, and the resulting normal appears smoother with its neighbours.

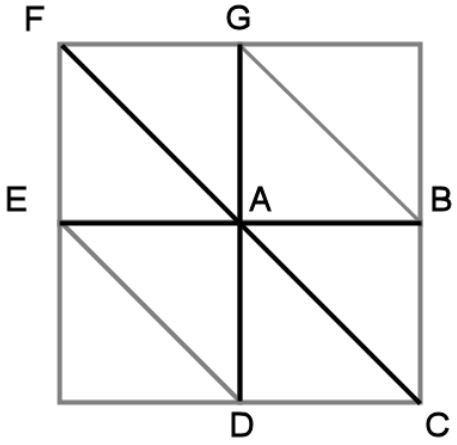



**Figure 3-19 Diagram of the points used in smoothed normal calculation**

The above diagram shows the triangulation for a 3x3 set of heights, creating a 2x2 set of triangle pairs. In order to calculate a smooth normal for A, the heights of A – G must be considered. The equation for calculating a normal for a triangle is the same as above, except this is repeated for 6 separate triangles: **GAB BAC CAD DAE EAF** and **GAF.** This results in 6 normals, which are then summed and normalised to result in 1 averaged normal.

$$\boldsymbol{n} = \left( \sum_{\substack{x=A \\ y=B \\ z=C}}^{G} (\boldsymbol{y} - \boldsymbol{x}) \times (\boldsymbol{z} - \boldsymbol{x}) \right)$$

**Equation 3 Sum of cross products for points A through G**

The result of the lightmap is quite pronounced when we compare the normal-calculated version with the lightmapped version.

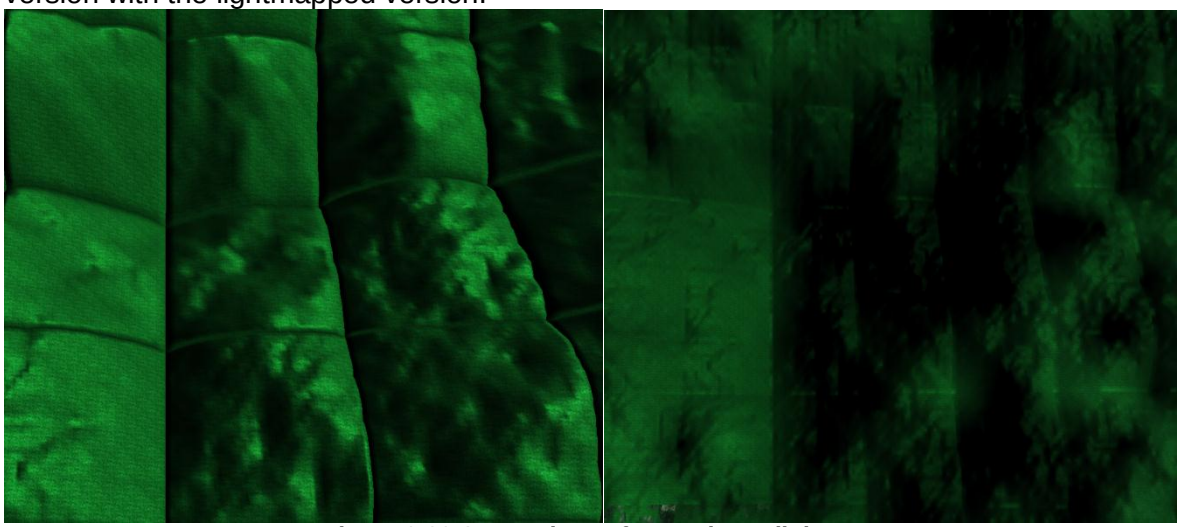


**Figure 3-20 Comparison of normals vs. lightmaps**

The normal-lit shot on the left has continuity errors at every block edge, but the important aspect is the image quality within the blocks, it looks quite jagged and undetailed. The lightmapped version on the right appears to have lots more detail, and also does not pop-in like the normal-lit version does.

### 3.3.3 Lightmap Continuity

Having assembled splatmaps and lightmaps for each TerrainBlock, the landscape now has texture and shading associated with it. During development, a visual artefact manifested whereby the seams between blocks were obvious due to differences in the lightmaps at the edges of the blocks.

The normal averaging algorithm needs the heightmap points to all sides of the vertex normal being calculated. This means that when the top left normal of a block is being calculated, the algorithm assumes that the surface continues in the direction the next vertex points to.

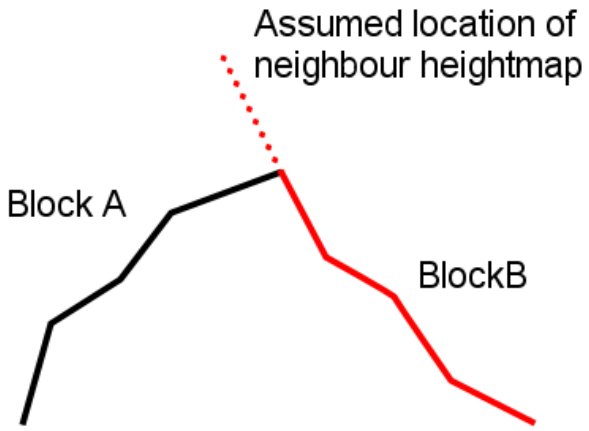



**Figure 3-21 Diagram of false assumptions made in lightmap calculation**

This shows a 2D cross section of the border between two TerrainBlocks. When calculating the left-most normal of Block B, the algorithm assumes the block to the left of it has its next height in the same plane as Block B's. This leads to a discontinuity in the shading that is obvious to the viewer.
For a block to have its lightmap fully generated, it needs to have data about the blocks around it, which might not have been generated yet.

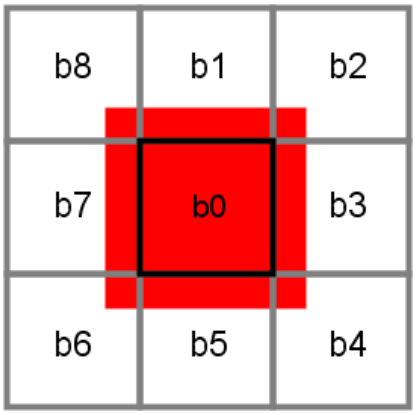



**Figure 3-22 Diagram of area of heightmap needed to smoothly generate lightmap for b0**

In order for the central block **b0** to have its full lightmap created, it needs all the data highlighted in red, so requires the heightmaps for **b1** to **b8** to have been generated. This will be referred to as a **Neighbour Set**.
The first approach to solve this problem was to create the lightmap as usual, but perform a correcting pass to modify the normals. Whenever a new TerrainBlock was created, it would check against nearby blocks to see if it completes a neighbour set. If so, the central block is given pointers to the normal maps of the nearby blocks, and the outside ring of normals is recalculated by smoothing current normals with the ones from neighbour maps.
This approach improved the problem, by smoothing the discontinuity between lightmap edges. However, this wasn't quite enough, because smoothing the normals wasn't mathematically identical as calculating it directly from the heightmap. This is because the underlying assumption about where the heightmap values were when calculating the lightmap the first time round is still present in the final data, although less apparent.
The second approach was to share neighbour heightmaps instead of normal maps, and feed that into the normal map calculation for individual blocks. This method delayed creation of the lightmap until the neighbour set was present. If the block was drawn at this point, it would be uniformly bright. When a neighbour set is completed, the centre TerrainBlock is sent pointers to the heightmaps of its neighbours, and the lightmap is calculated using the

neighbour heights as well as the block's current heightmap. This resulted in much smoother edges, as shown below.

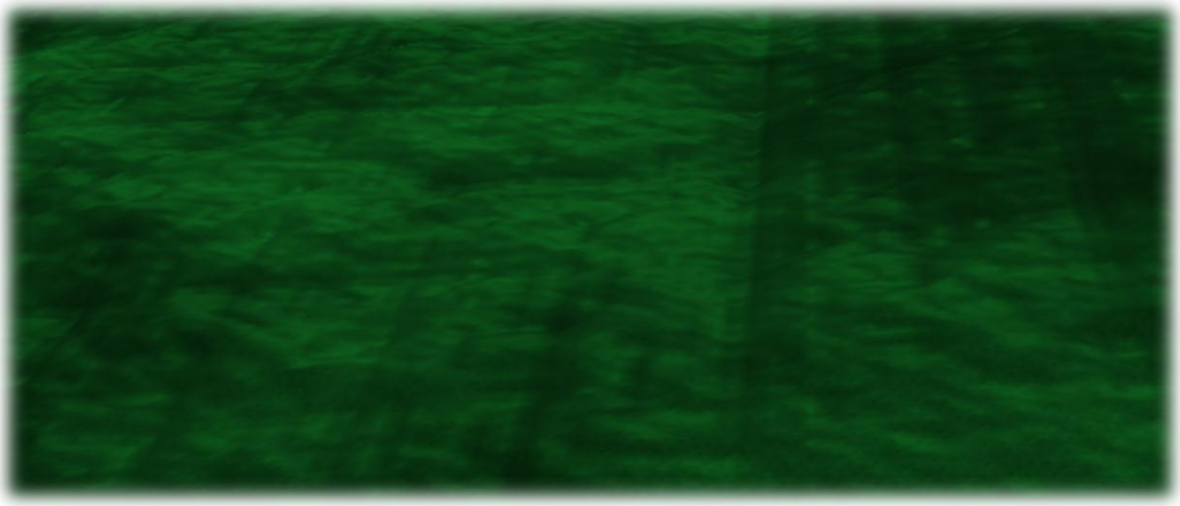


**Figure 3-23 Less apparent edges between blocks**

After completing this continuity correction, the edges between blocks were much improved, but there was still a problem. The lightmap only takes into account the few normals near any given vertex, so although there aren't any hard edges, the lightmap colour can change quite quickly, leading to lines across the terrain where there are harsh valleys created by the fractal terrain generation algorithm. The other problem with the current lightmap implementation is if a smooth surface, such as a sand dune is created. Because the underlying block is flat, the resulting desert looks like this:

**Figure 3-24 Flat-shaded desert looks square and low-res**

While the edges between blocks on the pixel level is smooth, it's still obvious that these surfaces are square and flat.
The solution that was implemented was to use bilinear filtering on the corner normals of the surface, so that the surface is shaded smoothly in both dimensions. A 1D analogue is shown below:

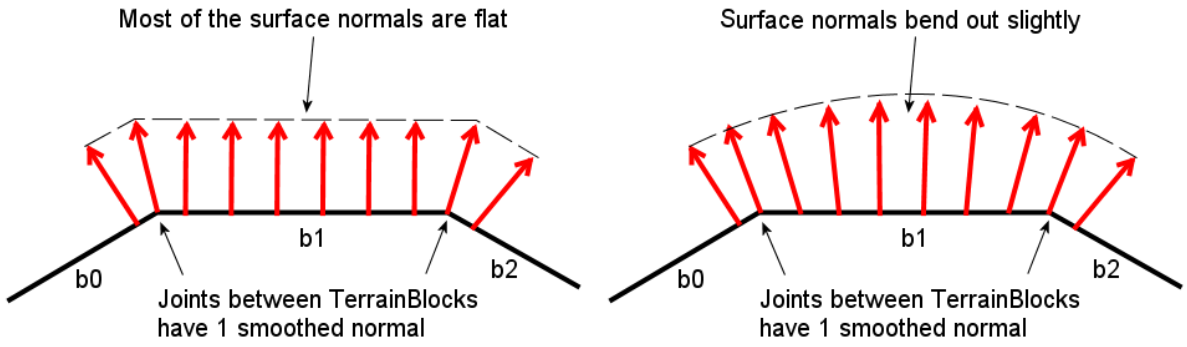



**Figure 3-25 Difference between flat surface normals and smoothed ones**

On the left is the unfiltered normals, where the normals between b1 and the other blocks are smoothed, but the rest of the surface is flat. On the right is the linear filtered version (extends to bilinear in the 2D case) where the normals curve out slightly, giving a smooth transition between the two three blocks. This makes the terrain look a lot smoother, as shown below:

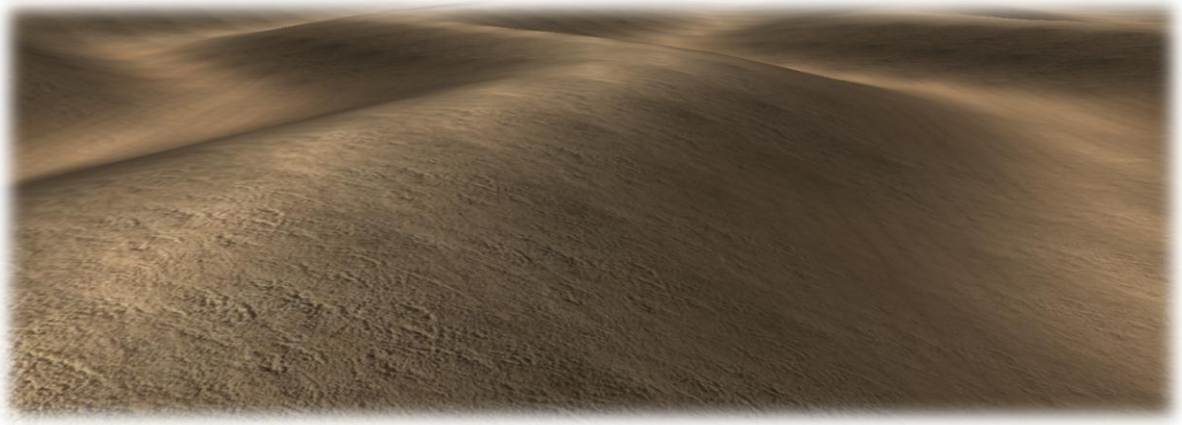


**Figure 3-26 Desert scene with smoothing applied**

The caveat here is that if bilinear filtering is used exclusively, it only takes into account the edge 4 normals, which means that the block is very smooth, but has no detail. In the current implementation, a blending factor is used, which combines the detail normals with the smooth interpolated normals. This means that for an entirely smooth area, like the desert, a factor of 0 is used, meaning the normals are entirely smoothed. For the hills and mountains, a factor of 0.5 can be used, which has a smoothing effect, but keeps high frequency detail.
With this implementation, the lines between TerrainBlocks are much less obvious, and helps provide the illusion that the landscape is a continuous whole.

### 3.3.4 Fractal Terrain Generation with Diamond Square

As outlined in 2.3.1.4 and 2.4.1, the diamond square fractal generation algorithm is a process by which heightmap values are iteratively displaced from the outside in, using a diamond square pattern. When generating a heightmap for a TerrainBlock, a set of seed points are sent to the GenerateHeightmap function. If no blocks have been generated yet, then only the four corner points are set at a neutral height. If one or more of the sides of the TerrainBlock has a neighbouring block that's already generated, that side acts as the seeds for the new terrain block, so that each block matches up exactly with its neighbours.
The first step in the algorithm is to take the 4 corner values, and average them to get the centre value.

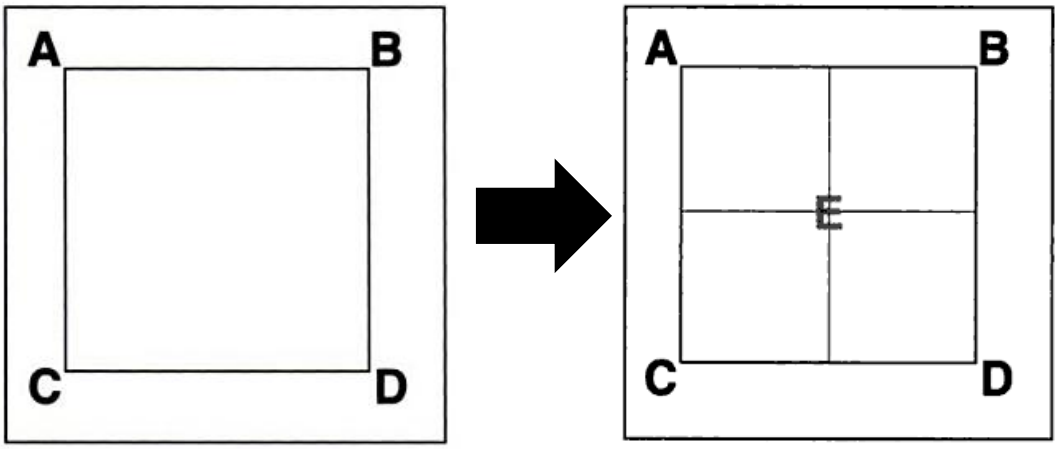



**Figure 3-27 Diamond Step (Polack T, 2002)**

Taking ABCD as the seed values, E is produced, this is the diamond step, as it averages the diagonals. A small random offset is added to E to add detail to the terrain, as iterations continue, the range of random numbers is reduced, which means that the initial passes of the algorithm give the overall shape of the terrain, while the later passes give the detail.

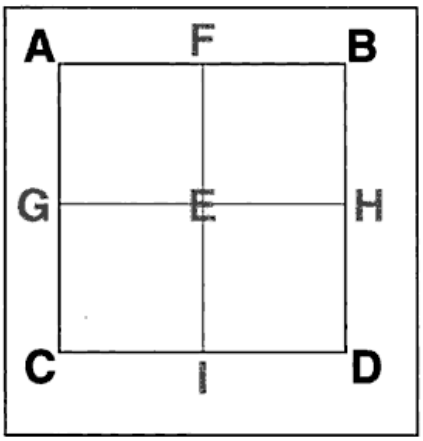



**Figure 3-28 Square Step (Polack T, 2002)**

The square step is now performed, where the midpoints are calculated between **ABCD**, and form the points **FGHI**. These points now complete the subdivision of this square into 4 smaller squares. The algorithm is then performed again on each of these squares again and again until the desired precision is achieved.
In this implementation of the algorithm, roughness and smoothing factors are used. When offsetting the heightmap points, the maximum possible offset would be 1 or -1, clipping the height to the top or bottom of the variable set. A good roughness to start with would be 0.4, which gives a well sized peak or trough in the middle of the TerrainBlock. If this roughness was kept constant throughout the algorithm, the resulting terrain would be very jagged and look very unnatural. Every iteration, the roughness factor is multiplied by a smoothing factor, which makes the roughness decrease over the iterations. A smoothing factor of 1 means no decrease in roughness and very jagged terrain, a smoothing factor of 0 means that everything but the first pass is infinitely smooth, and would end up with a pyramid shape, as all subsequent points just interpolate the initial 5 values. A smoothing factor of 0.5 would be a good start, meaning that the roughness halves each iteration, which keeps the overall shape of the terrain interesting, and keeps some detail in the lower frequency noise.

### 3.3.5 Infinite Scrolling

One of the challenges faced early on in the project was how to make the terrain scroll infinitely (or near infinite). It had already been decided that the terrain would be divided up into blocks to help with this, as it meant chunks of terrain could be swapped in and out when needed. The easiest way to depict a scrolling terrain would be to randomly generate new blocks when needed, and add them to the end of the terrain in the direction the camera is travelling. This would give the illusion of an infinite terrain, but if it was simply randomly generated, then when the camera returned to a particular point on the terrain, it would have been generated differently, breaking the illusion, as the world would not be consistent. To solve this problem, each block is given an X and Y blockID, which are combined to form a seed for the random generator, which means terrain is the same when returning to the same physical spot. This unfortunately means that it isn't entirely infinite, because the integer format used only goes between -2,147,483,648 and 2,147,483,647 which totals about 4 billion (thousand million) blocks in each dimension, which is 16 quintillion (trillion) blocks in total. This seems close enough to infinite to give the appropriate illusion.

The second problem is one of precision. The camera is positioned using floating point values, which can have high precision or great range. This means that if a floating point becomes very large, it loses local precision. This would mean that for extreme positions in the world, the camera might move a bit erratically, which would break the illusion. In order to solve this problem, instead of moving the camera, the world (TerrainBlocks) are moved underneath the camera, and new blocks are swapped into new rows when the camera needs to appear moving in one direction.

This needs two functions, one to move all the TerrainBlocks by a small amount, to give smooth scrolling, and another to swap blocks in and out as required, to keep them central to the origin.

The MoveTerrain function takes two float values and moves all blocks by a small amount, and then tests to see if the blocks have been moved further than a block length. In this case, the physical block size is 256, so if the terrain moves 256 in one direction, a new row needs to be swapped in.

```
if(terrainOffsetX > 256){
	terrainOffsetX -= 256;
	MoveBlocks(-1,0);
}
if(terrainOffsetX < 0){
	terrainOffsetX += 256;
	MoveBlocks(1,0);
}
```

The MoveBlocks function takes two integers and moves the currentBlockX and Y position by these values. Then the Update function will remove the unneeded TerrainBlocks, and swap in the new blocks when they've been generated.

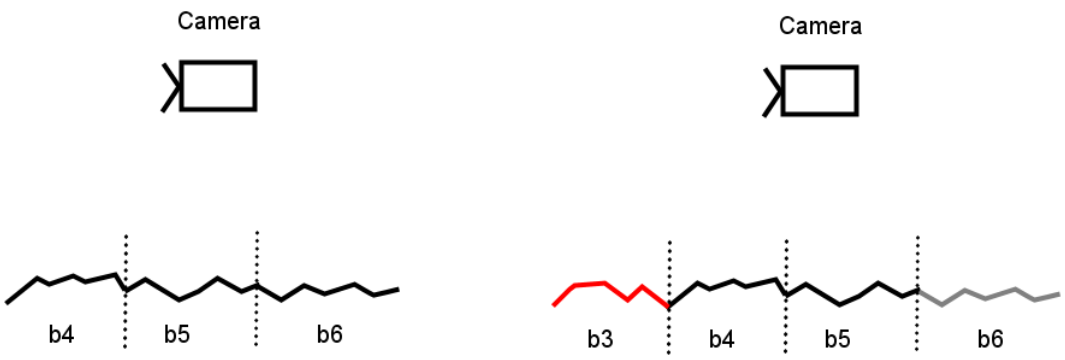


**Figure 3-29 Diagram of new block swapping in when camera moves**

In this example, b5 is the central block, with b4 and b6 also displayed. The MoveTerrain function moves all of these blocks to the right simultaneously until a new row of blocks are needed. At this point, b6 is no longer needed, and so will be swapped out. B3 however, is needed, so will be swapped in on the next update. This continues almost infinitely, giving the illusion of a complete world that the camera is moving through.

### 3.3.6 Super Blocks

Towards the final quarter of development time, it was decided that the landscape as a whole looked too boring. Although the individual TerrainBlocks were fractally generated, their seed points were still on a flat plane. This meant that the landscape was just homogenous looking lumps, as shown below.

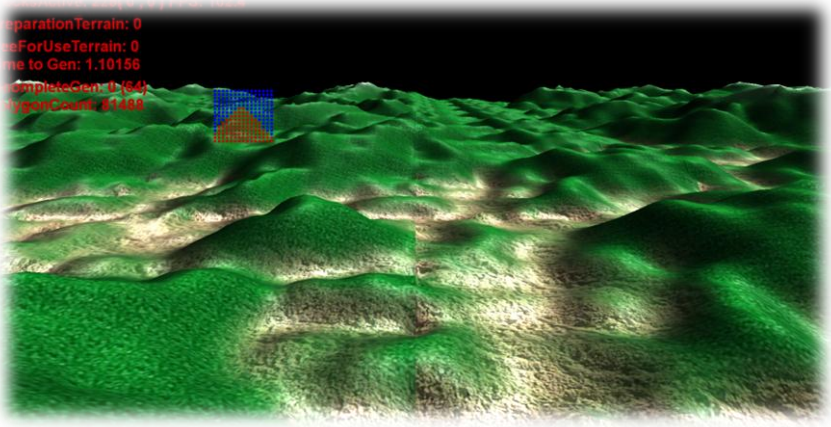


**Figure 3-30 Homogenous looking terrain**

In order to solve this problem, SuperBlocks were invented. The concept behind these would be that a super block would contain the seed data points for a 64x64 area of TerrainBlocks. This would allow a SuperBlock to describe the overall shape of a portion of terrain, such as the mountains and valleys, and the generation algorithm would seed each TerrainBlock with these points, and the TerrainBlock would fill out the details.

To fulfil the fractal aim of the project, the shape of the SuperBlock is determined by a Diamond Square algorithm, just like in the TerrainBlocks. When creating a new SuperBlock, this fractal algorithm is run, generating a 129x129 set of points, which are mapped to 64x64 TerrainBlocks. Each mapping is done using a 'BlockDefinition' which is a struct the SuperBlock uses to store seeding data about a TerrainBlock before it is generated. Each BlockDefinition stores 9 seed heights, for the Top, Middle and Bottom rows of Left, Centre and Right columns. This means that 129 points are used per dimension for a 64 block map.

In addition to the seed points, the BlockDefinition stores the roughness, smoothing factor and the normal blending factor. This means that the SuperBlock can determine how rough or smooth the TerrainBlock geometry will be, along with how smoothed its lightmap would be. This allows the SuperBlock to have small areas of rocky outcrops, with the rest of the SuperBlock being smoother. This is useful for making the tops of volcanoes ridged, while keeping the rolling hills smooth.

In order to keep terrain outline data in memory, SuperBlocks scroll with the terrain as the camera moves, so 9 SuperBlocks are kept in memory, the one the camera is currently over, and the 8 around it (N,NE,E,SE,S,SW,W,NW as compass bearings). When the terrain moves over a SuperBlock boundary, a new row of SuperBlocks are generated, and replaces the old row, ensuring that the generation thread always has a SuperBlock to get BlockDefinitions from when creating TerrainBlock heightmaps.

### 3.3.7 Feature Spots

After making SuperBlocks to add large scale sweeping detail to the landscape, it was decided that some more variety could be used to break up the scene and keep it from looking too homogenous. To this end, the FeatureSpot class was created. A feature spot is an NxN grid of offset values which deform the terrain of a SuperBlock into a more interesting shape. For example, a 10x10 grid could be assigned to a lakebed, where the [5,5] array value is largely negative, and the value tends to 0 as the array co-ordinates tend to the edge. This is effectively a heightmap offset map, which is used to make the SuperBlock generated terrain move up or down to varying degrees to make lakebeds, volcanoes and flat areas for forests.

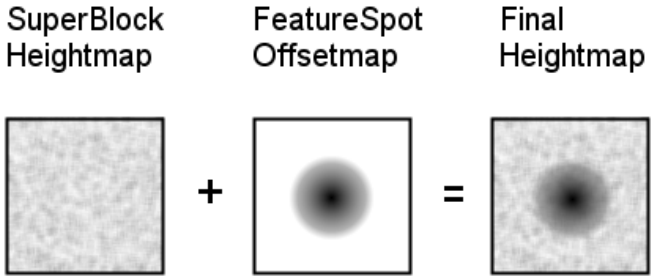



**Figure 3-31 Combination of heightmap and offsetmap**

FeatureSpots only affect the shape of the SuperBlock, and can't make new geometry. A new class is needed for the geometry detail that is often associated with FeatureSpots. For example, a lake FeatureSpot needs water geometry in it, otherwise it's just a hole. A forest would just be a flat space without the trees.

In order to accommodate this, the DetailObject class was created. This class is associated with the TerrainManager, and whenever the update method is called, it checks the current SuperBlock's array of FeatureSpots to see if detail needs to be added. If a new piece of detail has come into view, then its parameters are added to the detail object list, and is rendered the next time Render is called. A lot of the geometry used will just be duplicates, such as all lakes will be circular. In order to save on memory, the objects are loaded in as static pointers, associated with the DetailObject class, and then every instance of DetailObject that is rendered using the static geometry on the GPU. This is particularly important for trees, of which there can be hundreds on screen at any one time.

### 3.3.8 Trees and Geometry Instancing

One of the FeatureSpots mentioned previously was a forest – a flattened out area on which trees are rendered. The scale of these trees will have to be quite small to fit in with the rest of the world, so there will have to be large number of trees in a forest (about 80 is used). Drawing each of these trees individually would be a waste of CPU time, as each draw call uses CPU time, and each tree requires 3 passes (1 for the trunk texture, and 2 for the transparent leaf textures). This would result in 240 passes for a single forest, which would be a lot of work on the CPU, so the trees need to be batched together.

One possible solution would be to copy the vertex data for each tree into a large vertex buffer, which contains all 80 trees in different positions, allowing the forest to be drawn as one big object. This is known as static batching, and would cut down on draw calls, but be very wasteful of GPU memory, as it's storing 80 times the vertex data that is needed.

Thankfully, there is a method introduced in DirectX 9.0 called the Geometry Instancing API, which is designed for this very purpose. The API changes the way a shader receives information by sending two vertex buffers. The first is a normal vertex buffer containing the geometry of the object to be rendered. The second contains data about every instance, and the data represents variables such as position and colour.

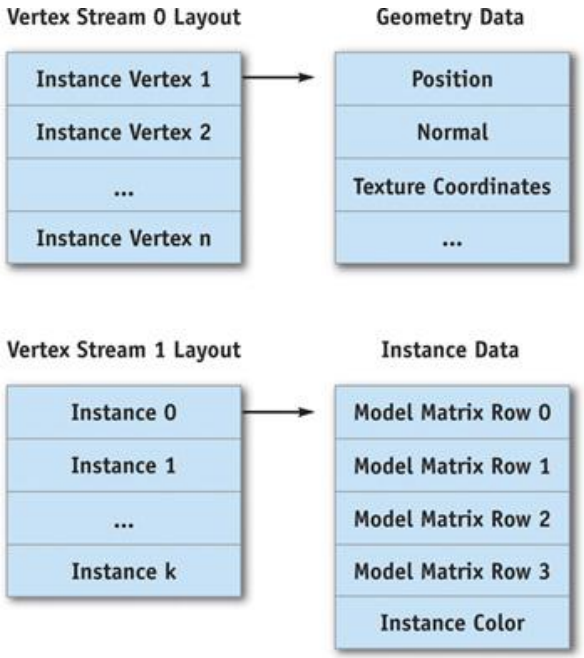



**Figure 3-32 Vertex Buffer Layout for the Instancing API (Carucci F, 2005)**

Using this system, 80 trees can be drawn in just 3 draw calls, one for each texture. However, an improvement can be made. By using an extra variable in the vertex buffer it can identify what texture should be used. In order to do this, the vertex buffer for the object is carefully ordered so that all vertexes using the same texture are grouped together, and the colours of the vertex are set to red, green or blue to tell the shader which texture to use. This is then fed into the instancing API as usual, and one pass is made to draw all 80 trees fully.

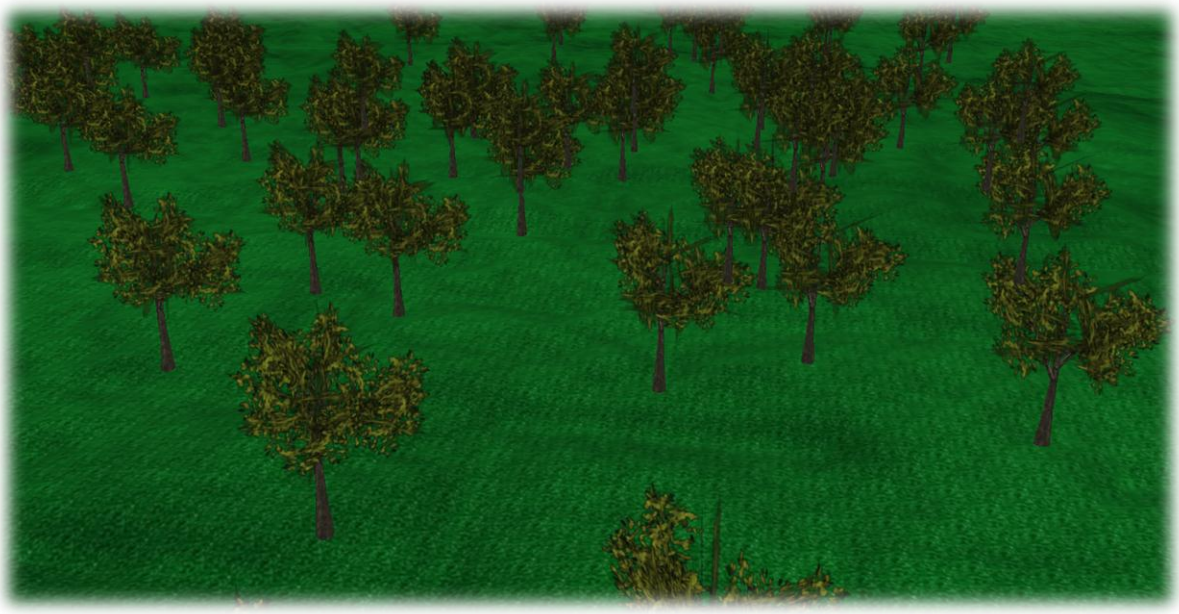


**Figure 3-33 Trees rendered in the final program**

### 3.3.9 TerrainBlockGroups and Batching

Interactive graphically intensive applications, such as this one, are often both GPU and CPU intensive. In the case of this project, the GPU needs to be capable of rendering large areas of terrain in a quick and efficient manner, complete with shaders and extra detail spots. The CPU on the other hand is busy with generating the geometry to be sent to the GPU, along

with the splatmaps and lightmaps, as well as coordinating threads and organising the TerrainBlock arrays. This means that the CPU needs to be freed up as much as possible to make the generation run quickly. With DirectX, whenever a Draw Call is made, it uses a chunk of processor time to coordinate it. Usually, the CPU can draw tens of thousands of these per second, but if framerate is considered, this gives only a few hundred to a thousand per frame if 60FPS is desired. In order to keep the draw calls down, objects can be grouped up into batches, and drawn as one object. In order to do this, TerrainBlockGroups are used. These objects take over from the TerrainBlocks when creating geometry, and instead of creating one triangle strip per TerrainBlock, it can create one triangle strip per 4x4 group of blocks, meaning that the number of draw calls is reduced by 16. The question is quickly raised: Why 4x4? Why not draw all the blocks as one call?
Firstly, if all the blocks were grouped together, whenever a new row of blocks were added, the TerrainBlockGroup would have to be updated, and regularly updating a large vertex buffer is slow, and it would also take the processor time to re-strip the triangles.
The other issue is one of culling. Drawing fewer polygons on screen makes rendering faster, so ideally, any polygon offscreen shouldn't be drawn. To do this, a culling frustum is used to determine what the camera is looking at. Initially, a custom implementation was chosen, based on code from (Fernandes A, 2010) but then was swapped out for the DirectX function D3DXPlaneDotCoord, which takes a plane and a position vector, and determines which side of the plane it is on. The DirectX alternative was used because the software implementation initially used wasn't 100% correct, so resulted in some odd results due to a code problem that wasn't identified.

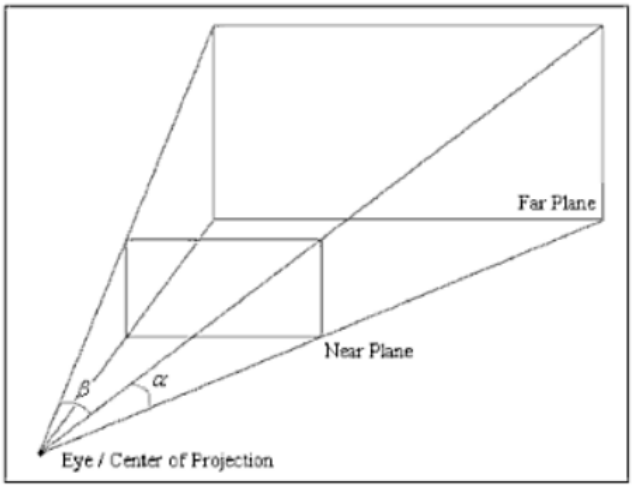



**Figure 3-34 Standard camera frustum (Luna F, 2006)**

As diagrammed above, the camera frustum consists of 6 planes, the near and far planes, as well as left, right, top and bottom frames. If a point is on the inside of all of these planes, then it is within the frustum and will be drawn onscreen, otherwise, it can be culled.

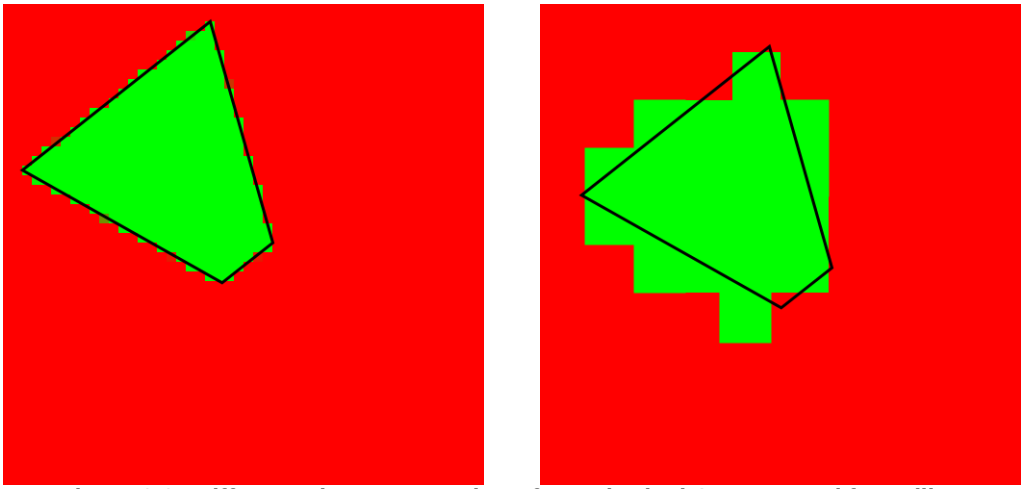


**Figure 3-35 Difference between number of TerrainBlockGroups used for culling**

On the left is a cull frustum on a lot of small TerrainBlockGroups, the green represents groups to be drawn, and the red represents those to be culled. The green fits well to the frustum, so there are not many polygons drawn that don't need to be. Conversely, in the right image, where a much smaller number of groups are used, there is a lot of wasted polygons, as the green often goes far past the culling frustum. If this was the only performance metric, then lots of TerrainBlockGroups would be good, and very few polygons would be wasted, but this would mean lots of draw calls. For this reason, a compromise between reducing draw calls and increasing culling efficiency is needed. Using 4x4 blocks turns out to be a good compromise, and ends up rendering about 70 draw calls, resulting in 90FPS average on test system A (see 6.1.1).
As mentioned previously, TerrainBlockGroups turn heightmap data into triangle strips. The heightmap data is provided in a grid array format, so the intuitive solution would be to use triangle lists, rather than strips. Triangle lists are where every triangle is formed by 3 vertexes, and they are listed 3 by 3 by 3 as a list of triangles. This means that some vertexes are repeated for grids like heightmaps. Triangle strips, on the other hand, start off with 3 vertexes for the first triangle, and every vertex after it forms a triangle with the previous three.

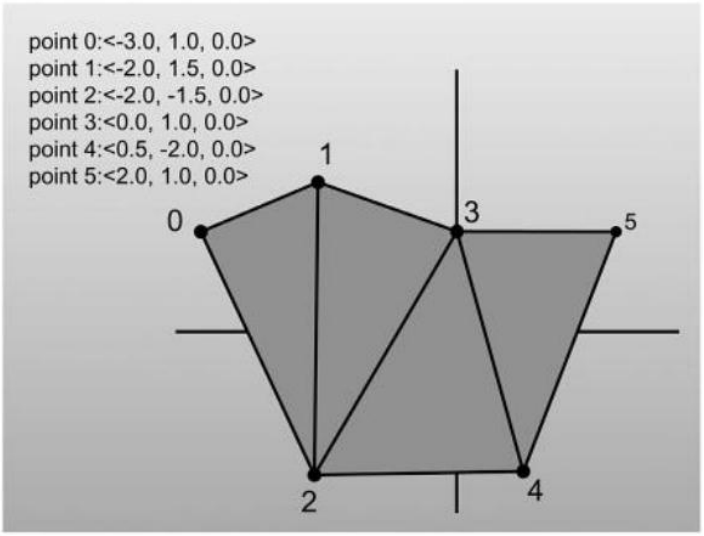



**Figure 3-36 Triangle Strip Layout (Walsh P, 2008)**

The image above consists of 4 triangles, which are described using 6 points in a triangle strip. In a triangle list it would take 12 points to describe. This saves a lot of GPU memory, and results in faster rendering. This method works very well for triangulating the first row of a

heightmap, but at the end of a row, it needs to reset to the left side and look at the next row, just like reading the text on a page of the book, there is a jump from the end of one line onto the next. If the triangle strip is just continued, a long thin triangle is drawn between every row, and ruins the visual effect. Splitting up the triangle strips for every row would end up in many more draw calls, slowing down the program. The most elegant solution is to use degenerate triangles. A degenerate triangle is a triangle with no area, which can be made by having a triangle with two points being identical. If you were to imagine drawing a triangle from points 0, 1 and 0 again on the image above, it would result in a line which has no area, so would not be drawn by the rasteriser.

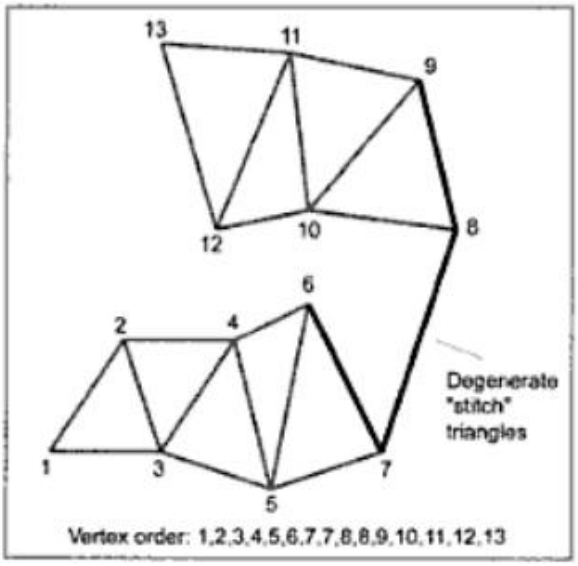



**Figure 3-37 Use of degenerate triangles to link separate strips (Fletcher D and Parberry I, 2002)**

As demonstrated above, the degenerate triangle results in a line between the two sets of strips, unifying them into a single triangle strip without artefacts. In order to do this, 4 degenerate triangles are used, so 4 more points are added in between the two strips. The last index of the first strip is repeated, followed by repeating the first index of the second strip twice, then the second index. The triangle strip can then continue as normal. (Reilly M, 2002)

### 3.3.10 Multithreading

One of the most challenging problems faced in this project was efficiently threading the application to generate geometry in the background, while rendering and updating in the foreground. The solution to this is to use multithreading, and have the main thread render and update the TerrainBlocks which are ready to be drawn, and have a worker thread generate geometry and lightmaps for the blocks that will be needed soon. This is an elegant solution, and makes use of the multicore processors of today. The caveat is that multithreading is very complex to orchestrate efficiently. If care isn't taken, then the application spends more time waiting on the other thread than would have been saved by having everything running in the same thread. The optimisation and experimentation for the threading system continued throughout the programming of this project.

Mutexes (Mutual Exclusion objects) are a system by which a mutex is flagged as acquired by one or other thread, and the thread which doesn't have the mutex must wait until the other thread is finished with it before continuing. The mutex acquisition operation is atomic thread safe, and so is used to flag pieces of code which act on data shared between the two threads. Some form of thread safety is needed because threads can run be running in parallel, so there is no guarantee that a piece of code will run or complete before another piece of code in a different thread. This is fine if the threads don't interact, but if there is data that needs to be passed between threads (such as TerrainBlock data) then thread safety needs to be used, otherwise undefined behaviour occurs.

The basic setup of the threads was to have two threads: UpdateDraw and GenerationThread. UpdateDraw used terrainBlockList to draw all the currently active blocks on screen and update them. GenerationThread used generationBlockList to store blocks that were in the process of being generated, and weren't on screen yet. The idea was that GenerationThread would be busy in the background, churning out geometry into the generationBlockList, and the UpdateDraw thread would swap in the generationBlocks into the TerrainBlocks and draw those.
The aim of using mutexes was to reduce the amount of time spent waiting for the other thread, so logically, the first solution was to have a mutex for every single TerrainBlock. This meant that if 100 blocks were to be drawn on screen, and another 40 were in the generation list, then 140 mutexes were to be used. This would mean that the only time a mutex would be needed simultaneously was when the same block needed to be drawn as was being generated. This turned out to be horribly inefficient, because mutexes are not very quick. The act of locking and unlocking a mutex can take up to 9000 processor cycles (Duffy J, 2006) and so having over 100 of these meant that and semblance of good performance was lost.
The second approach was to use 2 mutexes, one for each of the lists. This meant that while the GenerationThread was generating a generationBlock, the draw thread would be drawing the blocks in terrainBlockList, and no wait conditions would occur. The only place where both mutexes needed to be locked was where the generationBlocks were swapped into the TerrainBlockList for rendering. This was a marked improvement on the previous solution, but there was a problem. An extra lock was needed in the GenerationThread because when generating a new block, it needs to know about its neighbours to smooth the lightmaps and seed the geometry creation, as described in 3.3.3. This meant that only part of the GenerationThread was operating on the generationBlocks alone, it needed access to both mutexes periodically to sort out neighbouring blocks. While the performance was decent, it resulted in an uneven framerate. When the GenerationThread was idle, the framerate was very high, but when the camera moved into new terrain, and more blocks needed to be generated, the framerate took a huge hit, as it couldn't draw and generate simultaneously.
The final solution was to maintain a copy of all blocks in the generationBlocks list. This meant that if the grid of blocks on screen was 10x10, then the generationBlocksList would be 12x12, and would contain a copy of all the blocks on screen, and those being generated. This way, the GenerationThread never needs to lock the terrainBlocksList mutex, and drawing can go on simultaneously with generation. The consequence of this is that a lot of the TerrainBlocks are stored twice, wasting some RAM, but this is a reasonable price to pay for efficiency. The key part to making it work smoothly is the Update method. This is the only place where both mutexes need to be locked, and it is where generationBlocks are copied across to the terrainBlockList. If this blocked too often, then the framerate would take a hit. In order to make this fast, the Update method doesn't block, and merely waits for 1 millisecond to see if both mutexes are free. If the GenerationThread is busy with a generationBlock, then the Update method skips the block swap operation and goes back to rendering another frame. This results in a very smooth framerate, but at the expense of blocks that visually pop on screen in large chunks. This compromise was acceptable because realtime smooth framerate was deemed more important than pop-in.
A tool called Memory Validator (Software Verification Limited, 2010) was used during the multithreading testing to ensure that no memory leaks were present. This software would check for not only standard memory leaks, but for unsafe thread accesses, which made it a useful tool in ensuring that the code was thread safe.

### 3.3.11 Linear Fog

In order to disguise terrain geometry popping in at the horizon, a common approach is to use fog. In real life cases, fog is a phenomena caused by small particles or droplets in the air which scatter light and cause far away objects to be obscured by the fog. In rendering

processes, this can be simulated by tending far away objects to a particular colour. This is usually done by looking at the depth of a rendered pixel, and applying a colour offset to it depending on how far away it is.


**Figure 3-38 Example of linear fog in a sample application from ShaderX2 (Nuebel M, 2003)**

In the case of linear fog, the fog is applied according to the following equation:

$$f = \frac{Z_{fogEnd} - Z_{depth}}{Z_{fogEnd} - Z_{fogStart}}$$

**Equation 4 Linear fog equation from ShaderX2 (Nuebel M, 2003)**

This allows the start and end of the fog to be determined, which is useful in this case, as the aim is just to obscure the distance where the terrainBlocks will pop in without making the rest of the terrain difficult to see. The fog was implemented in the system using the DirectX API and its renderstate functions:

```
g_pDevice->SetRenderState(D3DRS_FOGTABLEMODE, Mode);
g_pDevice->SetRenderState(D3DRS_FOGSTART, *(DWORD *)(&Start));
g_pDevice->SetRenderState(D3DRS_FOGEND,   *(DWORD *)(&End));
```

As provided by Microsoft on the Pixel Fog page of MSDN (MSDN, 2011).
In action, the fog effect adds a great deal of realism to the scene.

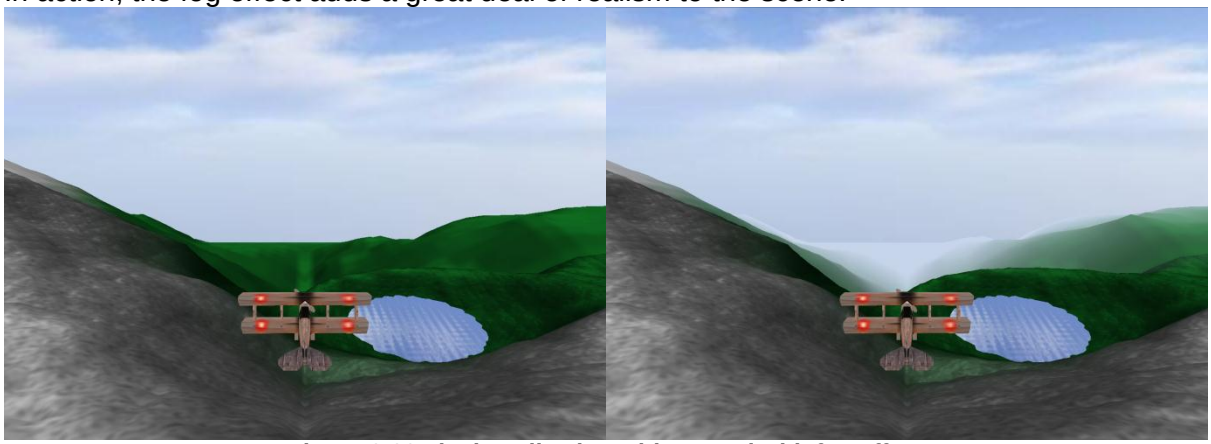


**Figure 3-39 Final application without and with fog effect**

### 3.3.12 Water Rendering

Another aesthetic addition which was added toward the end of development was water. The water in this simulation is limited to circular lakes which are placed at FeatureSpots, the detail of the water is implemented through a shader. More complicated solutions, such as form fitting lakes which fill in parts of the terrain geometry, or river simulation and water sources were not attempted due to time constraints and that they were outside the main focus of the simulation. One performance benefit to having all the lakes circular is that a single vertex buffer can be used for all lakes, saving video memory.
The shader used to simulate the water surface is a reflective shader based off a cube map texture of the sky box. A procedural offset to the normal is applied using trigonometry functions to create a ripple effect, and a specular component is added to give the water a brighter appearance.

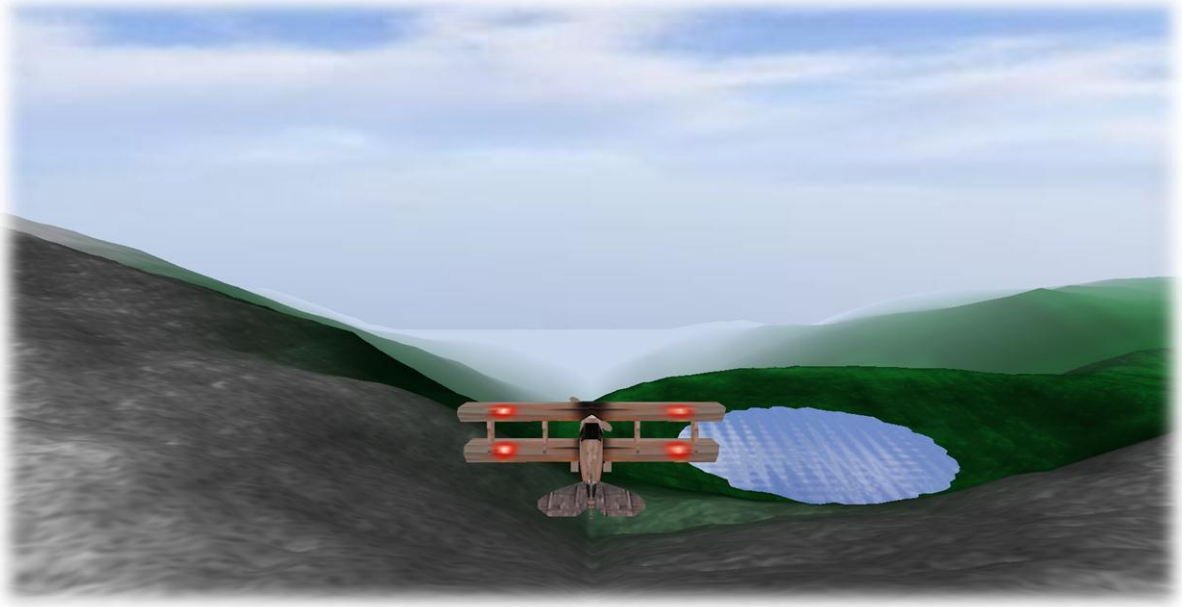


**Figure 3-40 Water effects in full scene render**

The techniques used to implement the water shader were inspired heavily by Ben Humphrey's implementation (Humphrey B, 2005)

### 3.3.13 Airplane Simulation

The airplane simulation aspect of the project did not require physical accuracy, so a simple model of acceleration and velocity were used to simulate the movement of the plane. The user only has three controls: Accelerate, Turn Left, Turn Right. The height of the plane is controlled automatically to keep it above the terrain surface. The airplane class hold variables for the plane's position, velocity and acceleration. The acceleration is applied to the position using the following formula, taken from (Li Q, 2011):

$$P(t) - P_0 = \int_0^t v(s)ds$$

$$P(t) = P_0 + v_0 t + \frac{1}{2}at^2$$

**Equation 5 Constant Acceleration formula (Li Q, 2011)**

And the velocity is also updated according to:

$$v(t) = v_0 + a_0 t$$

**Equation 6 Velocity formula (Li Q, 2011)**

This allows the airplane's forward velocity to be controlled realistically. The turn controls are implemented in a similar way, where a 'Turn Left' command accelerates the plane's rotation in that direction, and when no command is applied, the acceleration is quickly dampened to zero. This makes the plane feel as though it is 'banking' into turns as a real plane would, and makes the turns feel less rigid, without making the plane feel out of control.
The height of the plane is achieved by finding the SuperBlock that the plane is currently over, and calculating the height that the underlying terrain is at. This height (plus a small amount to keep the plane out of the ground) is classified as the minimum height the plane can be at. If the plane's current height is less than the minimum height, then the plane is set to the minimum height, to ensure no intersection with the terrain occurs. If the plane is above the minimum height however, the plane's height tends towards that height over a couple of seconds, which means that the plane floats down off of high mountains, giving a feeling of smooth flight, rather than being stuck rigidly to the terrain.

## 3.4 System Testing

During the development of the application, most testing was done as required to ensure a new feature was functioning correctly. Whenever a bug was found, a note of it was made, and it would be fixed at a later date. Bugs of negligible importance would be further down the list and would be fixed later (or not at all if time constraints were a problem). Some examples of bugs that were found during development were:

- Black lightmap edges, which were caused my incorrect ordering of generation, which meant that a lightmap would be generated without all its neighbours.
- Resizing or minimising the window results in a crash. Some DirectX resources needed to be re-instantiated in this case.
- TerrainBlocks didn't match up. Seeding from neighbours had been incorrectly implemented, and more checking needed to be done to ensure that blocks lined up.
- TerrainBlockGroups had gaps between different LODs. Different levels of detail meant that gaps appeared on boundaries. This was fixed by clipping higher res maps to lower res boundaries. The groups were large enough that this didn't affect visual quality much.
- Lake detail jutted out of mountains or were placed under the terrain. This was fixed by getting the heights directly from the SuperBlock, rather than estimating it with an equation.
- Crashes on non-dev machine. Inter thread memory leaks were occurring, meaning undefined behaviour, which was accentuated on other machines. This was fixed by properly employing mutexes.

The second form of testing was to test the software on different hardware platforms, to ensure compatibility and test performance. The specifications of the test systems used are available in 6.1.

| Hardware Used | Idle FPS | Generating FPS |
|---|---|---|
| Test System A | 105 | 90 |
| Test System B | 110 | 75 |
| | | |

# 4 Critical Evaluation

## 4.1 Project Achievements

Overall, the project has gone very well, and everything got finished on time, but some elements of polish were left out. The Time Plan was kept to quite closely, but needed some tweaking at the interim stage. Some of the tasks were started slightly earlier than expected

and lasted longer than expected. A large amount of time was put in on the part of the developer to ensure that the project was feature complete by the Easter break period. Although very time consuming, it has resulted in a very impressive project that meets all of the core goals set out initially.

Core Aims

- Render a 3D terrain
  - *Must be realtime (above 25FPS)* – **Complete**, the final application runs at well over 60FPS consistently on test system A (see 6.1.1)
  - *Must scroll as to appear larger than what is currently onscreen* – **Complete**, the world is scrolled under the camera, and new blocks are generated to ensure the illusion of a never ending landscape
  - *Base generation of terrain must incorporate a fractal algorithm* – **Complete**, this is achieved on two levels. The SuperBlock uses a fractal algorithm to generate the overall shape of the terrain, and the TerrainBlocks use a fractal algorithm to generate the detail in the lightmap and geometry.
  - *Must be textured, ideally with some form of shader* – **Complete**, texture splatting is employed in a shader to produce smoothly textured terrain.
- *Control a plane flying over the landscapes* – **Complete**, the user has basic control over a plane which flies above the landscape as it scrolls underneath it.

Secondary Aims:

- Additional Terrain Detail
  - *Include Trees on the terrain* – **Complete**, some FeatureSpots are forests, with many trees placed on the terrain.
  - *Include Lakes and/or Rivers on the terrain* – **Complete**, some FeatureSpots are lakes, which are simple, circular lakes. No rivers were incorporated.
- *Make the landscape scroll infinitely (if possible)* – **Complete**, the landscape scrolls almost infinitely, and certainly gives the illusion of an infinite landscape, as content is generated as the plane flies in a new direction.
- *Have the application scale graphics settings based on hardware available* – **Incomplete**, the software doesn't scale automatically to the hardware because there aren't graphical scale options incorporated into the application.

The only goal that wasn't met was graphics scaling to hardware. This wasn't achieved because of restricted time towards the end of the project, and that underlying code structure only allows for small amounts of graphical scaling.
The other, related shortcoming of the application is the lack of terrain size scaling. The prototypes started off with a few large TerrainBlocks, and then they grew smaller and more numerous until performance was being impacted, so they were grouped up into bigger blocks again. If a system had been designed where TerrainBlocks could vary in size, this issue could have been resolved in a better manner.
Also, a lot of time was spent optimising the generation algorithms, and in the end, the logic was consolidated down by about half. At one point, when a block was generated, it attempted to be swapped into the drawing list, if not, it went on a list to be attempted later, and then during the update method, swapped in. This was a waste of code, as the swap in code could be used just once in Update for a simpler and more effective method.

The big achievements of this project are the infinite scrolling, something that has been seen rarely in high graphical intensity applications. The smoothness of the threading algorithm is

also a particularly impressive achievement for a student project, although the speed of terrain generation could certainly be improved. The generation of blocks is a little slow on today's systems, and while there is room for optimisation, it is partly due to the complex nature of calculating new content in real time which is very demanding on today's machines. In a few years' time, this approach will be more viable, particularly when processors become more and more parallel, and something needs to be found for each of the processors to do. Another impressive point is the graphical fidelity of the project, the lightmaps provide a very smooth and detailed surface, and the texture splatting is very smooth, resulting in an aesthetically pleasing scene.

## 4.2 Further Development

There are a number of improvements that could be made to the project to improve it further.

- Improved terrain generation speed -
  The biggest visual problem is the pop-in that detailed lightmaps make. If the efficiency of the Generation algorithms were improved, this could give a more appealing result, this could be done by using lower resolution lightmaps, improving search algorithms, or optimising the quadtree currently in place.
- Fix tree transparency -
  Trees are not drawn in back to front order, so at certain angles, some transparency artefacts are present. These were not fixed in time due to trees being a non-core aim, and would have been left until other bugs had been fixed.
- Scalable TerrainBlock size -
  Refactoring some of the early code to allow terrain blocks to sample from the SuperBlock in a non-hardcoded way would allow better performance optimisation.
- Graphical Scaling -
  Allowing the lightmap quality to change and adding in multiple versions of shaders would allow the graphics to scale with the computer hardware.
- Greater variety of terrain types -
  If more texture sets and fractal parameters had been defined, then a greater variety of terrain could be achieved.
- Greater variety of FeatureSpots -
  The FeatureSpot class is quite versatile, and it would be good to add some more variety to them, including types such as Oasis, Rock Outcrop, Village/Houses or maybe fields/parks.
- Lockless programming –
  In order to speed up the multithreading aspect of the program, a possible solution would be lockless programming, which is a method of programming which allows thread to interact without locking mutexes or entering into critical sections. This was investigated briefly, but it was determined that it would require an overhaul of the application that would cost too much time. (Dawson B, 2003)

## 4.3 Personal Reflection

The author thinks that plenty of personal learning progress has been made as a result of this project. Learning about HLSL shaders and their integration into DirectX has been achieved, and good knowledge of the fundamental capabilities of shaders have been realised. Multithreading is another area that was little known about before undertaking the project, and while not mastered, a good deal of the basic efficiency dangers are now understood. The author also learnt a lot about DirectX bottlenecks and efficiency considerations such as draw calls and triangle strips. There weren't any huge problems managing a project of this size, but the code isn't as reusable as it could be, and that is another learning aspect. More in-depth knowledge about professional white-paper authoring and Harvard referencing has also been attained.

# 5 Bibliography

*Asirvatham A and Hoppe H, 2005, Terrain Rendering Using GPU-Based Geometry Clipmaps in Pharr M and Fernando R, GPU Gems 2, Massachusetts: Addison-Wesley, ch 2*

*Bloom C, 2000, Terrain Texture Compositing by Blending in the Frame-Buffer [online], Available: http://www.cbloom.com/3d/techdocs/splatting.txt [Accessed 28 December 2010]*

*de Boer W, 2000, Fast Terrain Rendering Using Geometrical MipMapping [online], Available: http://www.flipcode.com/archives/article_geomipmaps.pdf [Accessed 16 October 2010]*

*Burkardt J, 2001, MATLAB Graphics [online], Available: http://orion.math.iastate.edu/burkardt/papers/matlab_graphics/matlab_graphics.html [Accessed 12 April 2011]*

*Carucci F, 2005, Inside Geometry Instancing in Pharr M and Fernando R, GPU Gems 2, Massachusetts: Addison-Wesley, ch 3*

*Dawson B, 2003, Lockless Programming Considerations for Xbox 360 and Microsoft Windows [online], Available: http://msdn.microsoft.com/en-us/library/ee418650%28v=vs.85%29.aspx [Accessed 23 April 2011]*

*Desussen O and Lintermann B, 2005, Modelling Terrain in Deussen O and Lintermann B, Digital Design Of Nature, X.Media Publishing, p114 - 115*

*Duchineau M, Wolinsky M, Sigeti D, Miller M, Aldrich C, Mineev-Weinstein M, 1997, ROAMing Terrain: Real-time Optimally Adapting Meshes [online], Available: https://graphics.llnl.gov/ROAM/roam.pdf [Accessed 16 October 2010]*

*Duffy J, 2006, Using concurrency for scalability, MSDN Magazine September 2006 [e-journal] Available Through: Microsoft Developers Network Website [Accessed 12 April 2011]*

*Nuebel M, 2003, Introduction to Different Fog Effects in Engel W, ShaderX2: Introductions and Tutorials with DirectX 9.0, Wordware Publishing Inc, p151 - 179*

*Fernandes A, 2010, View Frustum Culling Tutorial [online], Available: http://www.lighthouse3d.com/opengl/viewfrustum/ [Accessed 12 April 2011]*

*Fletcher D and Parberry I, 2002, Triangle Meshes in Fletcher D and Parberry I, 3D math primer for graphics and game development, Jones and Bartlett Publishers, p325*

*Foxon C, 2010, Mountains in Canada [photograph] (Foxon C's Personal Collection)*

*Glasser N, 2005, Texture Splatting in Direct3D [online], Available: http://www.gamedev.net/reference/articles/article2238.asp [Accessed 28 December 2010]*

*Gosselin D, Sandler P, Mitchell J, 2005, Drawing a Crowd in Engel F, ShaderX3: Advanced rendering with DirectX and OpenGL, Cengage learning, p505*

*Henrichs S, 2010, Modelling Sand Dunes in 3DS Max [video online] Available: http://www.ronenbekerman.com/modeling-3d-sand-dunes-3ds-max/ [Accessed 23 April 2011]*

*Humphrey B, 2005, Realistic Water Rendering using Bump Mapping and Refraction [online], Available: http://www.gametutorials.com/Articles/RealisticWater.pdf [Accessed 23 April 2011]*

*Li Q, 2011, Physically Based Modelling – Kinematics, 08214 Simulation and 3D Graphics, University of Hull, unpublished*

*Luna F, 2006, The Rendering Pipeline in Luna F, Introduction to 3D game programming with DirectX 9.0c: a shader approach, Jones and Bartlett Publishers, p157*

*Martz P, 1997, Generating Random Fractal Terrain [online], Available: http://www.gameprogrammer.com/fractal.html [Accessed 16 October 2010]*

*MSDN, 2011, Pixel Fog [online], Available: http://msdn.microsoft.com/en-us/library/bb205332%28v=VS.85%29.aspx [Accessed 23 April 2011]*

*Olsen J, 2004, Realtime Procedural Terrain Generation, Thesis (PhD), University of Southern Denmark*

*Pelzer K, 2004, Rendering Countless Blades of Waving Grass in Fernando R, GPU Gems, Massachusetts: Addison-Wesley, ch 7*

*Polack T, 2002, Terrain 101 in Polack T, Focus on 3D terrain programming, Course Technology PTR, ch 2*

*Reilly M, 2002, Concatenating Triangle Strips [online], Available: http://www.gamedev.net/page/resources/_/reference/programming/sweet-snippets/concatenating-triangle-strips-r1871 [Accessed 12 April 2011]*

*Sestoft P, 2010, Numeric performance in C, C# and Java, IT University of Copenhagen Denmark*

*Shankel J, 2000, DeLoura M, Game Programming Gems Volume 1, Connecticut: Cengage Learning, p500-508*

*Software Verification Limited, 2010, Memory Validator, Version 5.00, (Software). [Downloaded 28 November 2010], Available: http://www.softwareverify.com/cpp/memory/index.html*

*Walsh P, 2008, 3D Math Foundations in Walsh P, Advanced 3D game programming with DirectX 10.0, Jones and Bartlett Publishers, p138*

*Wloka M, 2003, Batch Batch Batch What does it really mean?, Game Developers Conference 6-8 March 2003, San Jose, California*

# 6 Appendices

## 6.1 Appendix of Test System Specifications

### 6.1.1 Test System A

Processor: Intel Core-i5 2.40GHz
RAM: 4GB
Video: ATI Mobility Radeon HD 5650 1GB
Operating System: Windows 7
Resolution: 1920x1080

### 6.1.2 Test System B

Processor: AMD Turion X2 2.2Ghz
RAM: 4GB
Video: ATI Mobility Radeon HD 4530 512MB
Operating System: Windows Vista
Resolution: 1280x1024

## 6.2 User Guide

### 6.2.1 Installation

The main executable 'TerrainDX9.exe' is available in the TerrainDX9 folder, however there are three important dependencies that are required before it will run.

DirectX 9.0c compliant graphics card
Directx_feb2010_redist.exe
Vc_redist.exe

**This application has failed to start because MSVCR100.dll was not found. Re-installing the application may fix this problem.**
This problem refers to a missing dll required to run MSVisualStudio2010 applications. Running vc_redist.exe found on the CD will fix this problem.

**Unable to load d3dx9_xxx.dll**
This problem refers to a missing dll required to run DirectX applications made with the Feb 2010 version of the library.
Running Directx_feb2010_redist.exe found on the CD will fix this problem.

### 6.2.2 Operation

Using the application is very simple. Upon running, the landscape will display with an airplane in the centre of view. The user can control movement through using the W key to accelerate and the A and D keys to turn. The procedurally generated world can be explored like this and the plane will stay above the terrain currently flying over. The R and F keys control the height of the camera. P toggles between wireframe and solid view. U toggles a visualisation of the culling process (but slows down rendering slightly).

### 6.2.3 Shutdown

To shut down the application, click the close button at the top right of the window.

## 6.3 Credits and Sources

### 6.3.1 Software used

Microsoft Visual Studio 2010
Microsoft DirectX SDK
Windows 7, Vista
Microsoft Word 2010
Paint Shop Pro 8
Genetica Viewer
Mozilla Firefox 4

### 6.3.2 Sources of Textures and Models

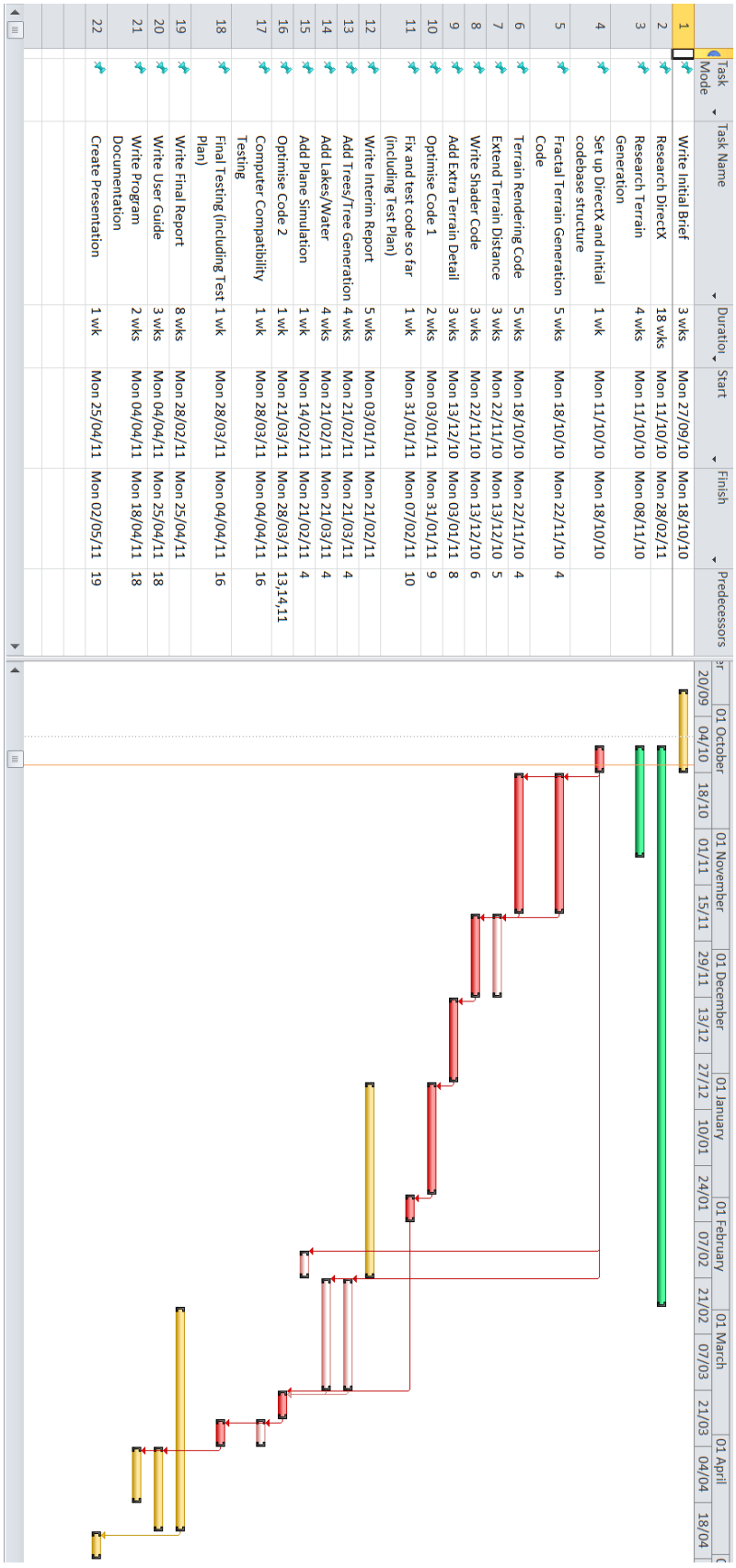
Mayang.com Texture Repository
Genetica Texture Creator and Viewer
Turbosquid.com
Microsoft DirectX Samples

### 6.3.3 Special Thanks

University of Hull
Darren McKie
Qingde Li
Friends and Family

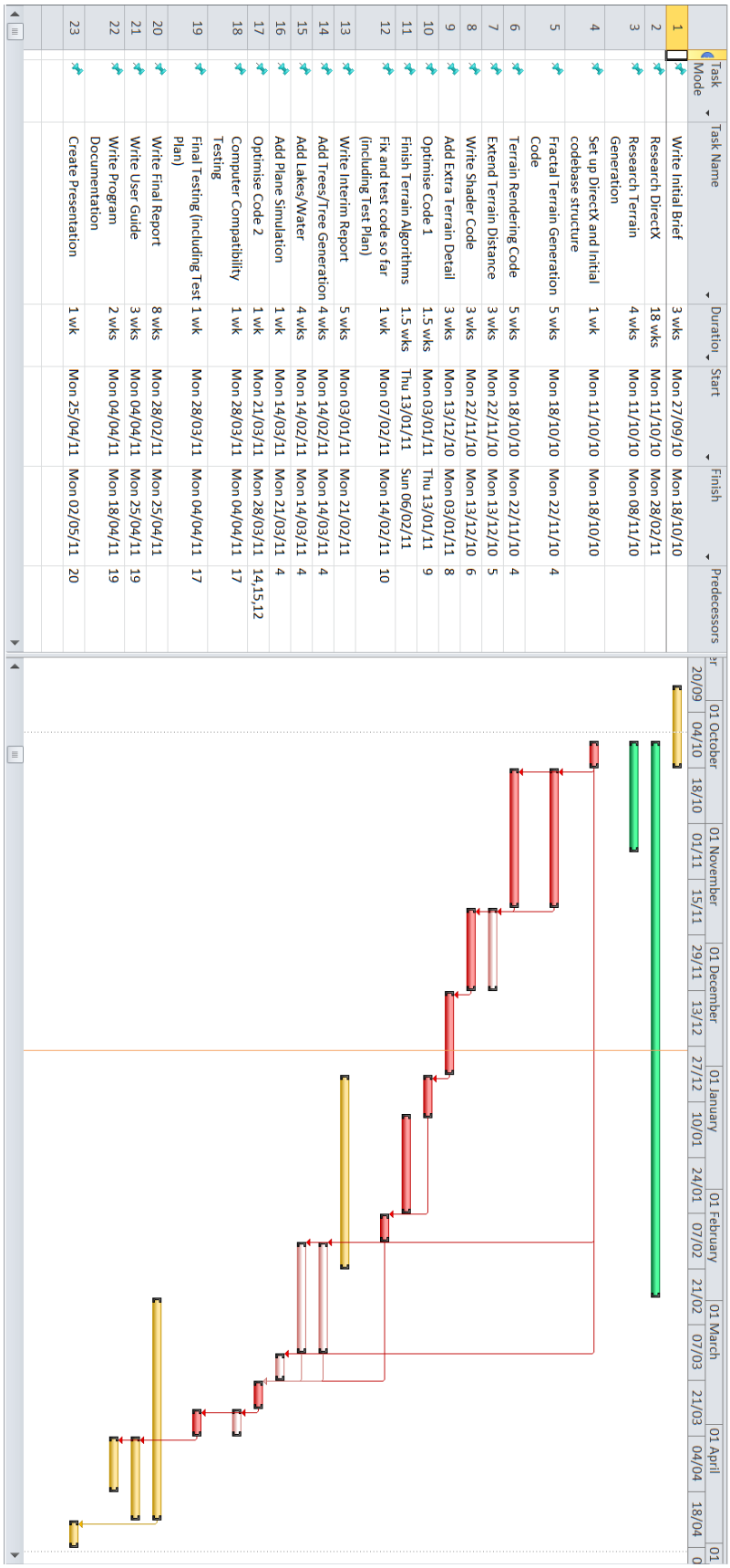
## 6.4 Timeplans

### 6.4.1 Initial



| | Task Mode | Task Name | Duration | Start | Finish | Predecessors |
|---|---|---|---|---|---|---|
| 1 | | Write Initial Brief | 3 wks | Mon 27/09/10 | Mon 18/10/10 | |
| 2 | | Research DirectX | 18 wks | Mon 11/10/10 | Mon 28/02/11 | |
| 3 | | Research Terrain Generation | 4 wks | Mon 11/10/10 | Mon 08/11/10 | |
| 4 | | Set up DirectX and Initial codebase structure | 1 wk | Mon 11/10/10 | Mon 18/10/10 | |
| 5 | | Fractal Terrain Generation Code | 5 wks | Mon 18/10/10 | Mon 22/11/10 | 4 |
| 6 | | Terrain Rendering Code | 5 wks | Mon 18/10/10 | Mon 22/11/10 | 4 |
| 7 | | Extend Terrain Distance | 3 wks | Mon 22/11/10 | Mon 13/12/10 | 5 |
| 8 | | Write Shader Code | 3 wks | Mon 22/11/10 | Mon 13/12/10 | 6 |
| 9 | | Add Extra Terrain Detail | 3 wks | Mon 13/12/10 | Mon 03/01/11 | 8 |
| 10 | | Optimise Code 1 | 2 wks | Mon 03/01/11 | Mon 31/01/11 | 9 |
| 11 | | Fix and test code so far (including Test Plan) | 1 wk | Mon 31/01/11 | Mon 07/02/11 | 10 |
| 12 | | Write Interim Report | 5 wks | Mon 03/01/11 | Mon 21/02/11 | |
| 13 | | Add Trees/Tree Generation | 4 wks | Mon 21/02/11 | Mon 21/03/11 | 4 |
| 14 | | Add Lakes/Water | 4 wks | Mon 21/02/11 | Mon 21/03/11 | 4 |
| 15 | | Add Plane Simulation | 1 wk | Mon 14/02/11 | Mon 21/02/11 | 4 |
| 16 | | Optimise Code 2 | 1 wk | Mon 21/03/11 | Mon 28/03/11 | 13,14,11 |
| 17 | | Computer Compatibility Testing | 1 wk | Mon 28/03/11 | Mon 04/04/11 | 16 |
| 18 | | Final Testing (including Test Plan) | 1 wk | Mon 28/03/11 | Mon 04/04/11 | 16 |
| 19 | | Write Final Report | 8 wks | Mon 28/02/11 | Mon 25/04/11 | |
| 20 | | Write User Guide | 3 wks | Mon 04/04/11 | Mon 25/04/11 | 18 |
| 21 | | Write Program Documentation | 2 wks | Mon 04/04/11 | Mon 18/04/11 | 18 |
| 22 | | Create Presentation | 1 wk | Mon 25/04/11 | Mon 02/05/11 | 19 |

## 6.4.2 Revised



| | Task Mode | Task Name | Duration | Start | Finish | Predecessors |
|---|---|---|---|---|---|---|
| 1 | | Write Initial Brief | 3 wks | Mon 27/09/10 | Mon 18/10/10 | |
| 2 | | Research DirectX | 18 wks | Mon 11/10/10 | Mon 28/02/11 | |
| 3 | | Research Terrain Generation | 4 wks | Mon 11/10/10 | Mon 08/11/10 | |
| 4 | | Set up DirectX and Initial codebase structure | 1 wk | Mon 11/10/10 | Mon 18/10/10 | |
| 5 | | Fractal Terrain Generation Code | 5 wks | Mon 18/10/10 | Mon 22/11/10 | 4 |
| 6 | | Terrain Rendering Code | 5 wks | Mon 18/10/10 | Mon 22/11/10 | 4 |
| 7 | | Extend Terrain Distance | 3 wks | Mon 22/11/10 | Mon 13/12/10 | 5 |
| 8 | | Write Shader Code | 3 wks | Mon 22/11/10 | Mon 13/12/10 | 6 |
| 9 | | Add Extra Terrain Detail | 3 wks | Mon 13/12/10 | Mon 03/01/11 | 8 |
| 10 | | Optimise Code 1 | 1.5 wks | Mon 03/01/11 | Thu 13/01/11 | 9 |
| 11 | | Finish Terrain Algorithms | 1.5 wks | Thu 13/01/11 | Sun 06/02/11 | |
| 12 | | Fix and test code so far (including Test Plan) | 1 wk | Mon 07/02/11 | Mon 14/02/11 | 10 |
| 13 | | Write Interim Report | 5 wks | Mon 03/01/11 | Mon 21/02/11 | |
| 14 | | Add Trees/Tree Generation | 4 wks | Mon 14/02/11 | Mon 14/03/11 | 4 |
| 15 | | Add Lakes/Water | 4 wks | Mon 14/02/11 | Mon 14/03/11 | 4 |
| 16 | | Add Plane Simulation | 1 wk | Mon 14/03/11 | Mon 21/03/11 | 4 |
| 17 | | Optimise Code 2 | 1 wk | Mon 21/03/11 | Mon 28/03/11 | 14,15,12 |
| 18 | | Computer Compatibility Testing | 1 wk | Mon 28/03/11 | Mon 04/04/11 | 17 |
| 19 | | Final Testing (including Test Plan) | 1 wk | Mon 28/03/11 | Mon 04/04/11 | 17 |
| 20 | | Write Final Report | 8 wks | Mon 28/02/11 | Mon 25/04/11 | |
| 21 | | Write User Guide | 3 wks | Mon 04/04/11 | Mon 25/04/11 | 19 |
| 22 | | Write Program Documentation | 2 wks | Mon 04/04/11 | Mon 18/04/11 | 19 |
| 23 | | Create Presentation | 1 wk | Mon 25/04/11 | Mon 02/05/11 | 20 |