Procedural Generation of Planetary Bodies

by

Dermot Hayes-McCoy, B.A.I., B.A.

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science in Computer Science

University of Dublin, Trinity College

September 2010

Declaration

I declare that the work described in this dissertation is, except where otherwise stated, entirely my own work, and has not been submitted as an exercise for a degree at this or any other university.

Dermot Hayes-McCoy

September 14, 2010

Permission to Lend and/or Copy

I agree that the Trinity College Library may lend or copy this dissertation upon request.

Dermot Hayes-McCoy

September 14, 2010

Procedural Generation of Planetary Bodies

Dermot Hayes-McCoy
University of Dublin, Trinity College, 2010

Supervisor: Micheal Mac An Airchinnigh

Procedural generation is a technique used in multimedia for the creation of a variety of its associated content. Such content may take the form of characters, images, landscapes, etc in both 2D and 3D applications and is usually created manually by highly skilled artists or designers. In contrast, procedural generation refers to the production of such content through automatic, algorithmic methods. Through automation of such expensive and time-consuming tasks procedural generation can save money and produce a more content-rich end product.

This report details a system capable of both generating and rendering in real-time procedural terrain in the form of planets. Although terrain is usually generated using a 2D heightmap image to represent elevation here a 3D heightmap is used. This results in a longer generation time but avoids many of the problems associated with mapping a 2D plane to a sphere and produces a much higher quality result. The heightmap itself is generated from Simplex Noise, using Fractional Brownian Motion to create a more realistic landscape topology. To create more heterogeneous terrain a series of

multifractal-based heightmaps are also used. The heightmap is not stored in memory as, depending on the size and number of planets to be generated, this could occupy considerable space. Instead, a value is generated for each input position on the fly, ensuring that computation time is determined by the degree of detail required on screen at any time rather than the size and extent of the terrain itself.

In order to display the terrain in real-time a Level of Detail (LOD) system is used to remove excess detail when necessary. This system uses an implementation of Geometry Clipmaps to achieve this, although heavily modified to accommodate spherical terrain. Improved performance is also achieved by utilising the GPU rather than the CPU in both the heightmap creation and LOD stage. As modern graphics hardware provides much greater computational power than the CPU for many tasks, offloading these stages to the GPU results in very good performance and leaves the CPU mostly free for potential future tasks.

Results show that these techniques are suitable for creating a number of procedurally generated planets in real-time, without consuming large amounts of memory. Frame rates of greater than 60 frames per second can be maintained whilst generating and displaying an earth-sized planet to a resolution of less than 1 metre per vertex. Furthermore, the various fractal techniques used are capable of creating heterogeneous terrain surfaces that provide a good approximation of realistic topography.

Contents

Abstract **iv**

List of Figures **viii**

Chapter 1 Introduction **1**

1.1 General Procedural Generation 2
1.2 Procedural Generation of Visuals 3
1.3 Applications 4
1.4 Noise Generation 6
1.5 Using Noise 8
1.6 Mesh Representation 10
1.7 Irregular Meshes 10
1.8 Regular Meshes 11
1.9 Semi-regular Meshes 12
1.10 Fractional Brownian Motion 12
1.11 Fractals / Multifractals 14
1.12 General LOD Systems 15
1.13 CLOD / DLOD 16
1.14 Visibility Culling 17

Chapter 2 State of the Art **19**

2.1 Realistic Terrain 19
2.2 Geometry Clipmaps 26
2.3 GPU-Based Geometry Clipmaps 27
2.4 Spherical Clipmaps 31

Chapter 3 Design **39**

Chapter 4 Implementation **41**

4.1 Overview 41
4.2 Terrain Class 42
4.3 BuildVB method 43
4.4 BuildIB method 45
4.5 Draw method 47
4.6 Terrain file 49
4.7 Noise file 52
4.8 fBm 53
4.9 MultiCascade 54
4.10 HybridMultiFractal 55
4.11 RidgedMultiFractal 57

Chapter 5 Evaluation **60**

5.1 Results 60
5.2 Potential Improvements 64

Chapter 6 Conclusions **66**

Bibliography **68**

List of Figures

1.1 A procedurally generated surface 2
1.2 A Heightmap representing a surface 7

2.1 The effect on fBm of H values ranging from 1 to 0. Taken from [1] . . . 20
2.2 Terrain from a Multifractal. Taken from [1] 24
2.3 The pyramid of Clipmap layers. Taken from [4] 26
2.4 Each layer in the clipmap is imperfectly centred. Taken from [4] 28
2.5 The detail layers are constructed from set templates. Taken from [4] . . 29
2.6 The levels in spherical clipmaps are concentric circles. Taken from [5] . 32
2.7 This shows the transformation of the vertex mesh. Taken from [5] . . . 33
2.8 This shows the mapping operations of the mesh. Taken from [5] 34

5.1 This shows the effect of fBm 61
5.2 This shows the effect of a multiplicative cascade 62
5.3 This shows the effect of the hybrid multifractal 63
5.4 This shows the effect of the ridged multifractal 63

Chapter 1

Introduction

This dissertation will be divided from this point onwards into six main sections. These are:

- Introduction.
- State of the Art.
- Design.
- Implementation.
- Evaluation.
- Conclusions.

In the Introduction section a general overview of the field of procedural generation will first be given. This will be followed by a section covering potential and current applications for procedural landscapes in a variety of industries. Finally it will deal with much of the necessary background techniques and information discussed in the later stages of this dissertation.

The State of the Art section will also cover information on procedural techniques, but will take a much more specific approach than that taken in the Introduction, going into detail on some previous work that is directly relevant to this project.

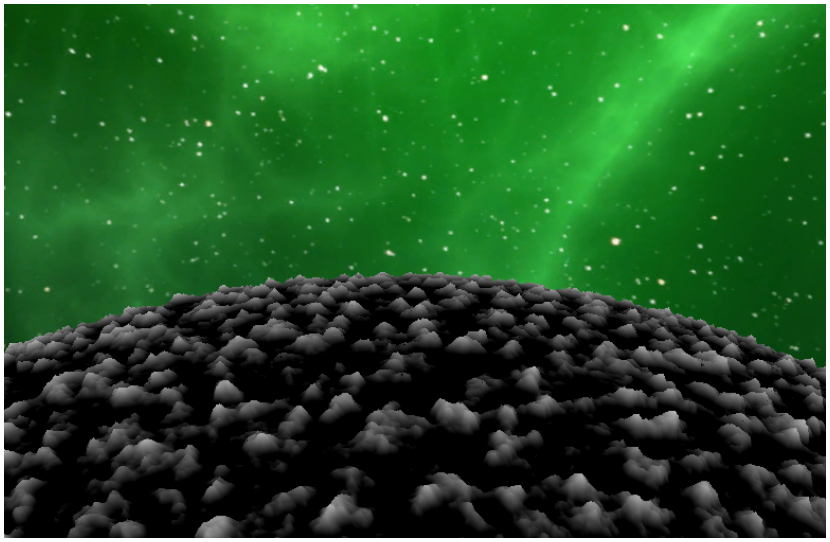

Figure 1.1: A procedurally generated surface

The Design section puts the information gained so far in context, outlining the form that the project will take and identifying what approaches will be used to tackle the various problems.

The Implementation section describes in more detail how the design plan was put into action. It discusses the architecture of the program and goes into detail on the manner in which specific aspects of the source code operate.

The Evaluation section describes the end results of this project. It examines both the positive and negative outcomes as well as describing potential future improvements.

The Conclusion section summarises the information of all of the preceding sections and puts the overall project and its results in context.

1.1 General Procedural Generation

The goal of procedural generation is to automate processes in the creation of a virtual environment that were formerly done by hand. These environments can be simple 2D paintings, 3D virtual worlds or even remain entirely abstract, existing only as internal

representations in a larger simulation. In this regard there are countless areas where procedural generation could be applied. This section will provide an overview of some of the most common uses, particularly those related to the topic of terrain generation.

The most common area is that of creating visual information for these environments, such as 2D backgrounds, 3D objects, etc. Other areas include procedural generation of music or the generation of storylines and plot. As the definition of procedural generation is very broad it can be claimed to include areas such as artificial intelligence, physics simulation, speech generation and animation. However, these are generally seen as entire fields in themselves so the title is usually applied only to cases outside of such well established domains.

Procedural generation is useful as, once the generation system is created, it can be utilised to continually produce content without the need for skilled content creators. Furthermore, large amounts of content can be created in a fraction of the time it would take to produce by hand. As the level of visual detail possible in media, particularly in film and video games, rises exponentially the traditional approach of manual creation becomes very costly in time and effort. Automating the task enables the construction of almost limitless amounts of content for a very small cost.

1.2 Procedural Generation of Visuals

Visual elements in multimedia applications that are generated using procedural techniques tend to fall within three main categories:

- Textures
- Objects
- World layouts.

Textures here refer to any 2D image created automatically. This can in some cases take the form of actual artistic pieces in the application (generally falling under the title of Artificial Creativity) or, more commonly, backdrops or tileable textures used to increase the believability and realism of the virtual environment. Tileable textures are images of repeating patterns, such as grass or brick, that can be placed regularly

over a surface to give the impression of detail. Backdrops also give the illusion of detail but are generally not repeated but maintained at a distance from the viewer to hide imperfections. Programs such as Terragen allow background images to be created and used in other applications.

Objects are 3D elements which can be created by algorithmic processes. As some objects can be quite unique they are often difficult to create computationally. The advantage of procedural techniques mostly lies in adding extra small scale detail that would be time consuming to do by hand. Furthermore, if a large quantity of similar objects is required in the scene then the use of multiple perfectly identical objects can be jarring and break the suspension of disbelief. To this end it is possible to create what is known as an 'Imperfect Factory', which is capable of producing many versions of the same object but with subtle differences in each to represent natural variation. Speedtree is a good example of this: it designs an infinite number of unique trees according to various parameters. The user can then place a large number of these trees over an area to create a convincing forest.

The final category of visual generation in common use is that of world layouts, and here lies the focus of this dissertation. A world layout is an entire virtual area within an application. It therefore can include many examples of textures and 3D objects, potentially even procedurally generated ones as mentioned above, but concerns itself with the macro-scale arrangement of elements to produce a convincing environment. It may describe the arrangement of a room, of a city or even an entire universe. Each potential type of layout would require its own algorithm, yet they can be broadly split into two types: man-made and natural layouts. Man-made layouts tend to favour regularity and hard lines whereas natural layouts are more uneven and often lack clear definition. This dissertation deals with the creation of planets in a geographic sense and therefore, falls under the latter category.

1.3 Applications

The potential applications for procedural terrain generation are varied but mostly fall within one of two categories:

- Entertainment

- Scientific / Training

The medium of entertainment is perhaps the one in which potential benefits can most easily be recognised. Many video games are requiring exponential increases in the quantities of terrain they use to represent their game worlds. This is especially true with the recent popularity of so-called open-world games, such as the Grand Theft Auto Series[25], Red Faction Guerilla[26], and Fallout 3[27]. These games require a large staff of artists and designers to create their environments, incurring substantial costs. In contrast, titles such as FUEL[28] and Infinity: Quest for Earth[29] make use of procedural terrain generation to synthesise landscapes of larger scales for lower cost. In film the expanding visual capabilities of the medium have created many of the same difficulties, to which the procedural solution can also be applied. Although not as widespread in the film industry procedural generation has been used to great effect in films, such as Inception[18]. Tron was perhaps one of the first instances of procedural generation used in film[6], and from it came the technique of perlin noise generation, described in more detail below.

There are a number of scientific applications for these techniques also, mostly falling within the area of astronomical, cartographic or geographic fields. These are also joined by simulation and training applications and often such programmes cover both categories. For instance the Virtual Airspace Simulation Technologies programme[19] is a joint NASA-FAA endeavour to vastly improve air transportation through both improved modelling and training programmes for both air traffic control and pilots. To this end the simulations need to accurately display large amounts of terrain of a spherical surface in order to appropriately simulate the goal environment.

Other potential uses include extra-planetary simulation, such as the Eagle Lander application[21] and the Mars Simulation Project[22]. Such simulations, or refinements thereof, could be used for planetary exploration and, although they usually operate on a predefined height representation, can use many of the same terrain display techniques as demonstrated in this project. It is, of course, quite conceivable that procedural height information would be required in addition to this, to represent unknown terrain or to quickly generate suitable environments for certain scenarios. Further examples exist in the form of maritime simulation for both dock and ship piloting training[30] as well as mineral exploration and oil drilling[20]. These simulators require extremely

large amounts of accurate terrain to be utilised and therefore could benefit greatly from a procedural approach. A number of applications alrady exist that create procedural terrain[15][16][17], however these applications do not focus on real-time generation of terrain, as out implementation does. Such programs are intended to create content on a purely offline basis.

1.4 Noise Generation

Pseudo-random number generators are often used as a base for many procedural techniques. The use of a given algorithm with random numbers as input should produce varied, and hopefully useful output. True random numbers would be undesirable as it would render it impossible to recreate a certain end result. Pseudo-random numbers enable the same result to be created so long as the seed value remains constant. This allows an element to be stored solely as a seed value, rather than a complete representation, saving space in memory but incurring greater computational expense.

Simple random number generation is usually too basic for many procedural applications. A further development of random numbers is procedurally generated noise. Noise can take the form of an $Ndimensional$ grid of pseudo-random numbers, which are independent of each other. A simple two dimensional grid of numbers can be interpreted as a two dimensional image, for example Figure 1.2, but the system can be easily extended into three or more dimensions. The exact interpretation of the numbers is left to the user: three dimensional noise could be utilised to generate three dimensional objects, such as clouds for instance. A fourth dimension could be added to act as time, resulting in an animated cloud that moves and changes shape. For each $N - dimensional$ input position supplied to the noise function it will produce a single value. An important aspect is that values can be obtained for specific positions as needed rather than obtaining values for every position an advance.

The manner in which these numbers are interpreted can change the character of the noise and produce large changes in the end result. Value Noise is often seen as the simplest form. In it each grid point possesses a fixed value and values for locations between the points are obtained through simple linear interpolation of the nearest points. A less obvious approach is to interpret the number at each point as a gradient rather than a final value in itself. The actual value at each grid point is always taken

Figure 1.2: A Heightmap representing a surface

to be zero with a gradient assigned randomly. The values at all locations between grid points can be non-zero and are determined by the gradients of the nearest points. It can be thought of as a grid of vectors, each representing a normal at that point. One can extend this in many ways, such as declaring that the number assigned to each grid point represents a combined value and gradient, as is the case with Value-Gradient Noise, or by doing away with a regular grid altogether, demonstrated in Sparse Convolution Noise. However, in most cases standard gradient noise is sufficient and it is this form which is used in one of the most popular types of noise: Perlin Noise[6].

The manner in which the system finds values between grid points is often the distinguishing feature between various types of noise. Perlin noise operates by finding a hypercube of grid points surrounding the input position and using a piece-wise hermite spline for interpolation of the query point. This means that interpolation has to occur between 2^N points (where N is the dimensionality of the noise) for each input position, with the result that generating an entire noise field can be computationally expense. For instance creating a three dimensional noise cube with 512 values per side would require $512^3 * 2^3 = 1,073,741,824$ interpolation calculations. It should be noted, however, that generating an entire noise field is often not necessary. Perlin noise is generated on a point-by-point basis, therefore noise values can be obtained for a series on N-dimensional coordinates without generating an entire N-dimensional area.

Simplex noise is an alternative to Perlin noise that offers lower computational com-

plexity at higher dimensions through the use of simplices rather than hypercubes[8]. Whereas a hypercube is effectively a square expanded to N dimensions a simplex takes the triangle as its base form and expands that to the required dimensionality. Simplices require a lower number of vertices to describe them than hypercubes. As stated above hypercubes are described by 2^N vertices whereas a simplex can by described by $N+1$. Simplex noise utilises these simplex vertices for interpolation with the result that in the above example only $512^3 * (3+1) = 536,870,912$ interpolation calculations need be made.

As gradient noise has a value of zero at each grid point its behaviour is, to a certain degree, predictable. Features will occur at a regular distance from each other and all values will always fall within certain bounds. This is generally seen as advantageous in procedural generation because it allows the user to create randomness yet control it through certain input parameters (such as the distance between grid points for example). The user also exerts control on the system in regards to the use of noise values and their treatment within the system. Controlled Random Processes (CRPs) such as these are important in content creation because users generally want to direct the process in some way. This provides an important distinction between procedural generation and entirely random generation. Even traditional painting on canvas can be seen as something of a CRP, because the painter uses the uncontrolled nature of the individual bristles to add detail to an overall brush-stroke that is controlled by the painter.

1.5 Using Noise

For this project noise generation is used to create a three-dimensional landscape, therefore the pattern of noise that is created needs to somehow represent the vertices of the final terrain. One easy way of doing this is to create two-dimensional noise and to treat it as a heightmap. A heightmap can be thought of as a top-down map of the terrain. The x and y positions in the noise field can directly map to the x and z positions of the terrain vertices, while the value at those points in the noise field corresponds to the height (y element) of the terrain vertex. In this way a flat $x-z$ plane of terrain vertices can be expanded into three dimensions by adding height values to each point, creating hills and valleys. The noise field can be thought of as a function: it takes as

inputs two points ($xandz$) and returns a single point (y) as output. One limitation of this approach is that only one height value can be obtained for each position vertex. This means that features such as overhangs become impossible, but in practice this is not a major issue. Realistic terrain contains very few natural overhangs is if such a feature is strictly required in a terrain simulation then it can be created by hand where needed. A number of sources exist to either create or supply heightmap data for other applications[13][14][15]. These can be used in place of a local procedurally generated version. Storing large heightmaps in memory can take up large amounts of memory and bandwidth, however.

Modifications of a plane can be accomplished using a 2D noise field but to modify a sphere a different approach is needed. One solution is to create a 3D noise field and pass as inputs the (x, y, z) co-ordinates of each vertex on that sphere[23]. As before, the noise function returns a single value as output. In two dimensions this represents a height value, which is simply the value along the y-axis. For the spherical case however height must be measured from the centre of the sphere to the vertex. It is clear, therefore, that if a new vertex position is obtained by adding height to the initial position, then these two points and the centre of the sphere are collinear. The new position can be found by taking a unit vector from the centre of the sphere in the direction of the initial position and multiplying it by the output of the noise function. In this case the output can be more accurately thought of as the distance from the centre rather than simply height.

As discussed previously, 3D noise generation is more expensive than 2D noise yet the 3D implementation mentioned above is not strictly necessary. It is possible to use a spherical coordinate system to represent the sphere in terms of only two values. The first is an inclination angle measured between the line from the centre to the pole and the line from the centre to the point in question. The second is an azimuth angle measured from a fixed reference position on a plane orthogonal to the line from the planet centre to its pole. In more common parlance these two values are referred to as latitude and longitude respectively. Inputting these two values into a standard 2D noise field will produce a height value for each vertex yet without substantial alteration this approach will result in problems.

These problems manifest themselves as visual artifacts both around the poles and along the line where the azimuth angle is zero. In the noise field values beyond a certain

distance from each other are completely independent. On a planar surface this works well as points that are far from each other in noise space are also far from each other in world space. When applied to a sphere however, points which are distant in noise space can be very close in world space. This is noticeable at the poles where points that have very different azimuth angles (distant in noise space) are located close by each other in world space. A similar problem lies in the fact that points with azimuth angles approaching 2π and those with azimuth angle 0 are also located next to each other in world space. If two dimensional noise is to be used to determine height for spherical surfaces then these problems need to be addressed or the final result will not look realistic.

1.6 Mesh Representation

Terrain is created from a mesh of vertices, onto which heightmaps are mapped to produce the desired terrain variation. In the case of planes and spheres it may seem trivial to create a suitable mesh yet its layout can affect the speed and accuracy of the final result. Layouts generally fall into one of three categories[31]:

- Irregular Meshes
- Regular Meshes
- Semi-regular Meshes

1.7 Irregular Meshes

An irregular mesh is one in which the position of its vertices are not restricted. This means that triangles can be created of varying size and shape so long as they don't overlap. The advantage of such an approach is that vertex density can be concentrated where it is needed most. Flat terrain, which has a relatively constant gradient, can be represented by only a few, large triangles, whereas terrain that is very rough, and therefore has a varying gradient, can use a larger number of small triangles. For a given triangle number of triangles irregular meshes can provide an optimal depiction

of the underlying terrain by saving detail for where it is most needed, unlike regular meshes described below.

There are a number of disadvantages to using irregular meshes. One of these is directly related to its potential advantage of using small triangles in high detail areas: namely that some triangles can become too small to be of any visual benefit. The extent to which this occurs depends on the underlying height information, but it can cause a significant problem, adding extremely small vertices that are of no benefit at the expense of detail in other, potentially more advantageous locations. Perhaps the most serious problem, however, is in the computational expense required to create them. The calculation of vertex locations requires significant overhead, and makes the operation of Level of Detail systems (described below) much more difficult. For this reason irregular meshes are usually only used in situations where the overall triangle processing power in the graphics device is very low compared to general computational power. This is not the case in most modern devices.

1.8 Regular Meshes

In contrast to the irregular variety regular meshes consist of evenly spaced vertices and therefore a defined distance between each. This produces a mesh of uniformly shaped triangles arranged in a grid. No modification is made to account for underlying terrain information and therefore every location on the grid is defined to the same detail. One advantage of such an approach is that heightmap data can be mapped to vertices very easily; in fact even a direct $1 : 1$ ratio of heightmap points to vertices can be used, eliminating interpolation calculations altogether. Also, due to the regularity of the data, large batches of triangles can be sent to the GPU with little or no computation, bringing about much faster rendering speeds. Finally, any calculations or modifications on the terrain are usually simplified due to its homogeneous nature. This includes real-time terrain alterations as well as Level of Detail operations and Visibility Culling (as described below)

1.9 Semi-regular Meshes

Due to the focus on accuracy in irregular meshes and on simplicity and low computational expense in regular ones a number of approaches exist which attempt to use aspects of both. Semi-regular meshes do not have evenly spaced vertices as is the case with regular meshes, but restrictions on their location and spacing do exist. They therefore attempt to increase the vertex density where needed, like irregular meshes, but also maintain some form of structure in order to simplify calculations. One example of this is by restricting all triangles to the shape of right-angled triangles. In this case the density of these triangles can be changed as needed but the layout can be calculated and stored easily, often in the form of tree based data structures.

Semi-regular meshes retain some of the detail of irregular meshes and some of the storage and computation efficiency of regular meshes. Traditionally two of their main disadvantages are that triangle construction requires a lot of CPU time and that passing batches of such triangles to the GPU is inefficient. This is not necessarily the case any more though when using modern graphics hardware, due to geometry shaders; these allow much of the grid detail to be created directly on the GPU and so alleviate both problems.

1.10 Fractional Brownian Motion

Using noise to directly produce height information does produce terrain but it is in no way indicative of the realistic terrain that this application aims to create. Terrain created straight from a gradient noise field often consists of a continuous series of peaks and valleys, which one would not often find in nature. As mentioned previously gradient noise is created from a grid of evenly spaced points. The frequency of these points is the same as the frequency of the terrain features in the final result. Real terrain does not possess such a set frequency in its features. For example a mountain range might have a frequency of a mountain on average every 1km, but it might have smaller dips and heights that would occur perhaps every 100m. On top of that are smaller rocks and clefts that occur on average every 10 metres and so on, right down to the smallest particles of soil. From this model we can see that as the frequency of the feature increases its associated effect on the final height decreases. Small rocks

and pebbles would occur frequently but their effect on the overall height at that point is minimal. In contrast mountains occur much less often but have a huge effect on the height of any points they are placed on. It is clear that the overall height at any given point is determined from the combination of each of the frequencies, obtaining the sum of the low-frequency, high-amplitude values all the way to the high-frequency, low-amplitude values. This results in the final terrain possessing detail at a number of frequencies, producing a much more believable look.

A pseudo-code implementation of this approach might have the following form:

```
for(int i = 0; i<octaves; i++)
{
     noiseSum = amplitude * Noise(sampleCoords);
     amplitudeSum += amplitude;
     amplitude *= gain;
     sampleCoords *= lacunarity
}
noiseSum /= amplitudeSum;
return noiseSum;
```

The meaning of each variable is as follows:

octaves: the number of different frequencies that are to be applied. Standard noise is just sampled at one frequency but this approach uses a number of different frequencies.
noiseSum: the final value that is to be returned. In this case where terrain is being determined it will represent the height value at the required point. Set to zero initially.
amplitude: the current scaling factor for the noise. This is used to weight noise at different frequencies to different degrees. Usually set to one initially.
amplitudeSum: a value indicating an ongoing sum of amplitude values used. Necessary for normalisation of final result. Set to zero initially.
noise(...): a standard gradient noise method that will return a value for given input coordinates
sampleCoords: the input coordinates for which this code will return a height.
gain: a value that controls how the amplitude is to be modified for each successive

octave.
lacunarity: a value that controls how the frequency of sampling is to be modified for each successive octave.

The division operation is simply to ensure that the values returned lie between 0 and 1. This enables a maximum height to be set if desired. Example values for summing over 8 frequencies going from low to high frequency might be:

octaves: 8
gain: 0.5
lacunarity: 2

A value of 2 is usual for the lacunarity variable. This will double the sampling frequency for each octave. Values less than 2 are to be avoided: they can suffer from recursive feedback as each noise field is added to a field similar to itself.

In the field of procedural generation the above method is claimed to generate Fractional Brownian Motion (fBm), a random walk process[1]. This is not strictly true as true fBm operates across all frequencies while this method only operates on a fixed number of frequencies equal to the number of octaves used. Using a high number of octaves however could be said to approximate the process. For the purposes of simplification though this report will refer to the approximation method detailed above as fractional brownian motion from this point onwards.

1.11 Fractals / Multifractals

Fractals are loosely defined as shapes or patterns that display the property of self-similarity[1]. This means that a part of the shape, when taken by itself, is similar to a reduced size version of the whole. As an fBm heightmap consists of standard noise at various scales it too can be considered a fractal. It should be noted that this self-similarity is not infinite but depends on the number of octaves used in the generation process. However infinite self-similarity is not a requirement for fractals and often occurs over only a small number of scales.

All terrain generated from the fBm method will have roughly the same character

throughout: continuous rolling hills over the entire surface. The fBm fractal should therefore be thought of as a monofractal due to this homogeneous nature. While this may be suitable for some areas a realistic terrain model should have natural variance. Some areas should be covered by flatlands, some mountainous. On some occasions the mountains should rise dramatically out of lowland terrain, on others the land should gradually rise up into hills and then mountains. Furthermore in some places the mountains themselves should be very rough and jagged, whereas in other locations the mountains, while still large, should be relatively smooth. In reality many of these characteristics are caused by other factors such as erosion and thermal action. Some procedural terrain generators simulate these effects with good results but visually appealing variation can be created through much simpler means, despite not being strictly accurate.

The use of multifractals is one such method; they are in contrast to the standard monofractals in that their behaviour varies strongly with location. A simple implementation would be to raise the final value to a power before returning it. This would have the effect of exacerbating the differences between high and low height values, giving smoother lowlands rising up to jagged peaks. Other solutions are to utilise a multiplicative cascade rather than an additive one as in fBm. This creates a stronger relationship between each octave and produces terrain that is heterogeneous over large areas. A more complex approach would be to change the height generation algorithm altogether, based on the input coordinates.

As no qualitative definition of good terrain exists any number of methodologies may be used to some effect. Creating a height generating algorithm is currently more of an art-form than science as even small changes to this algorithm usually bring about rather unexpected results. Thorough experimentation, rather than theory creation, is therefore usually the best approach.

1.12 General LOD Systems

In addition to creating terrain it is of course desirable to have some means of displaying it visually rather than maintaining it as an abstract construct. Large terrain sizes result in high computational cost, as each terrain vertex must be first calculated procedurally and then displayed on screen. Simultaneously generating and displaying large amounts

of high detail terrain becomes prohibitively expensive in real-time but many techniques exist to cut down on this cost with minimal degradation of the final image.

One solution to the problem is to cut down on the expense of height calculations; however an often easier approach is to use Level of Detail (LOD) techniques. LOD techniques involve decreasing the complexity of a 3D model by removing and merging vertices to allow for faster calculation. Ordinarily this would have a detrimental effect on the image but through careful analysis this can be reduced to a minimum, if not eliminated entirely. For instance, by comparing the position of the camera to that of the vertices the density of vertices in any particular area of screen space can be determined. If this density is higher than the actual screen resolution then the effect of many of these vertices is zero and they can safely be removed. Other LOD schemes use techniques such as varying levels of importance for different objects as removal criteria or take into account the current speed of the camera, reasoning that the viewer cannot sufficiently see as much detail when moving rapidly.

1.13 CLOD / DLOD

There are two main methods of implementing LOD functionality: Continuous Level of Detail (CLOD) methods and Discrete Level of Detail (DLOD) methods. CLOD systems work by adding enough individual details to a vertex mesh to satisfy some criterion, such as a minimum density of vertices in screen space. CLOD techniques can be difficult to implement because calculations need to be made to determine the number of vertices to be added and how to modify the existing mesh to accommodate them. DLOD techniques, on the other hand, use a set number of meshes of increasing detail to achieve a similar effect. Rather than continually modifying the mesh a DLOD system will swap its entire mesh a version of lower or higher detail as the need arises. As is the case with CLOD DLOD systems use a threshold factor to determine when to replace the mesh. This threshold is usually much larger than CLOD thresholds though due to the limited number of mesh versions that exist. Due to this fact DLOD techniques can be susceptible to undesirable visual effects such as popping, where the change from one mesh to another is clearly noticeable to the user. The individual meshes in a DLOD system can be either be pre-computed and stored in memory or generated on the fly as needed.

1.14 Visibility Culling

Whereas LOD systems operate by simplifying the detail of meshes other approaches improve performance by removing entire meshes or sections of meshes entirely. This is generally in cases where they do not contribute at all to the final image and so drawing them is unnecessary. One of the simplest approaches is Backface Culling, which removes triangles that face away from the camera. As it is handled automatically by most graphics libraries it won't be discussed here. Another approach used commonly is View Frustum Culling, the aim of which is to remove any vertices outside the camera's frustum of vision. Due to the camera's limited field of view, often only a subset of all vertices in the world are capable of being projected to screen space. Any others outside of this are calculated and drawn, but do not not appear on screen. This is clearly unnecessary and culling vertices outside the view frustum can bring about great performance increases. For example, if the view frustum was set at $90\,^{\circ}$ in both the x and y directions, then the entire world space would be six times the size of the view frustum space.

Culling entire objects that lie outside the view frustum is relatively easy. Likewise culling small objects that lie partly in the view frustum can be reduced to a simple binary choice based on a threshold. In the case of large objects, such as terrain for example, culling the entire object based on a threshold is unsuitable. Often on a small part of this object will be visible on screen at one time, but can take up a lot of the screen space, so passing a threshold would result in the object suddenly popping into existence on screen, which can be jarring. Furthermore, when the threshold has been passed and the object is drawn, a large proportion of it may be outside the frustum, negating much of the potential benefit.

It is therefore usual to split up the object into many smaller sections. These sections are evaluated independently of each other, allowing only part of the whole object to be drawn. Obviously the manner in which the objects are split is important. Sections that are too large have much the same problems as drawing the entire object. On the other hand sections that are very small bring about the most efficient culling but introduce a large overhead of computational cost due to the fact that each needs to be evaluated individually. Further complications can result in the splitting process itself as splitting irregular meshes can be very difficult to accomplish computationally.

For this reason regular meshes have a distinct advantage because they can be divided simply and efficiently into distinct blocks.

Chapter 2

State of the Art

This section provides a more detailed analysis than the more general introduction and covers some of the papers and other resources that are particularly relevant in the implementation of this project.

2.1 Realistic Terrain

In Texturing and Modeling - A Procedural Approach[1] a variety of fractal-based methods are discussed for the purposes of generating terrain. Most of these start by using fBm as a baseline and developing improvements on it. Their initial fBm method also offers some improvements over the method listed previously, adding a new variable, the fractal input parameter, to modify the amplitude for each octave. This parameter controls the character of the fractal and is generally used with values ranging from 0 to1. Values of 1 maintain a relatively smooth surface, as in the previous implementation. As this value decreases, however, the height surface returned becomes rougher, each point deviating more from its neighbours. The surface eventually approaches white noise as each vertex point is more and more independent of its surroundings. Figure 2.1 shows the effect of decreasing H values on a one dimensional surface.

A further improvement to the algorithm is that of partial, or fractional, octaves. These occur when the value of octaves used is a non-integer value and involve adding a partial octave to the standard integer octaves. The partial octave is created as normal but its result is scaled by the non-integer element of the octaves value, meaning that

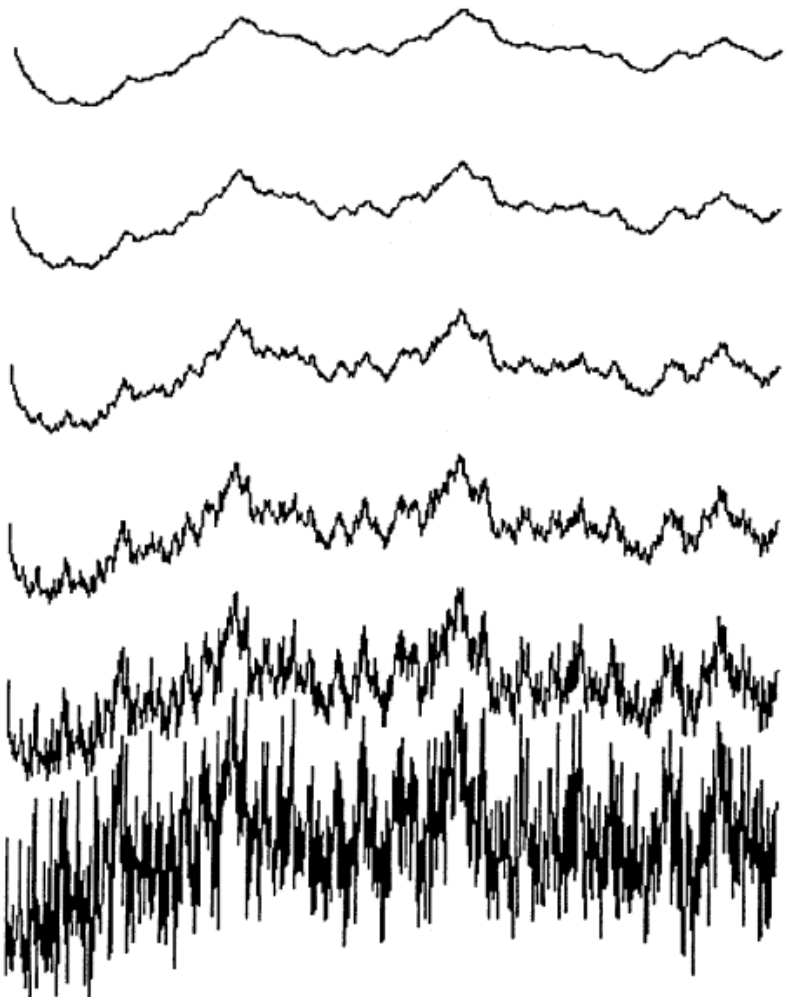

Figure 2.1: The effect on fBm of H values ranging from 1 to 0. Taken from [1]

its values will have less of an effect. The pseudo code for the improved fBm algorithm is as follows:

```
value = 0.0;

// Main loop of fractal construction
for(i = 0; i < octaves; i++)
{
     value += Noise(coordinates) * lacunarity ^ (- H * i);
     coordinates *= lacunarity;
}

// Find the non-integer remainder
remainder = octaves  toInteger(octaves);
if (remainder > 0.0)
{
     // Add partial octave
     value += remainder * Noise(coordinates) * lacunarity ^ (- H * i);
}

return value;
```

Here the value variable contains the sum of all noise generated so far and H is the fractal input parameter. Much as before, the code cycles through each octave adding its contribution to the final value. However, in this case there are no amplitude or gain values. Instead such functionality is subsumed into the H variable, and both it and the lacunarity value determine the scaling for the noise at each octave. It can be seen that as the H value increases the scaling factor for each additional octave in the loop is lessened, whereas if the H value was set as 1, it would have the same effect as a gain of 0.5 (provided the lacunarity stayed at its standard value of 2, of course). The addition of H enables easy parametrisation of surface roughness, unlike the more unpredictable variables it replaced. Note, however, that this approach does not set bounds on the

maximum height generated, which could therefore vary hugely. The same approach of dividing by the total of all amplitude values can be used here nonetheless, and should return a height value between 0 and 1.

Fractals can be described in terms of their dimensionality much like euclidean spaces. Dimensionality in such spaces takes the form of an integer value, such as a value of 1 for a line or 2 for a plane, etc. Fractal dimensions (as defined by the author) operate in the same way, in that a space with a fractal dimension of 2 is locally a plane and a fractal dimension of 3 is locally a three dimensional space. However it can also include real valued numbers, such as 2.31, for example. The closer the value is to an integer dimension the closer it represents that dimension, so a fractional value close to 2 would be a slightly roughened plane. As the value increased this plane would become rougher and rougher, expanding to fill a 3 dimensional area as the value approached 3. Usually these areas are deliberately bounded rather than infinite, so that, for example, a two dimensional plane representing terrain does not go on forever or a height value does not go beyond a maximum point. Usually the non-integer element is taken by itself to represent the character of the fractal, referred to as the fractal increment. For example, a space with a fractal dimension of 2.31 would have a fractal increment of 0.31. The H value described above is directly related to the fractal increment of the surface, as:

$$FractalIncrement = 1 - H$$

It is clear therefore why the H value is used in fractal generating techniques: it changes the roughness of the fractal by modifying its dimensionality. For extremely rough terrain at H values of near 0 the surface is simply expanding to take up the whole three dimensions within its bounding limits. H values are set between 0 and 1 as this forms a fractal increment from 0 to 1 also. It is important to note that the local definition of dimensionality in this case is not the same as the global dimensionality. A curved surface could have a fractal increment of 0, yet it is clearly not two dimensional. The author defines local dimension to exist at any scale, so, in the above example, if the curve is scaled up by a sufficient large factor the same sized area on the surface approaches a plane. If the fractal increment was non-zero this would not be the case.

The partial octaves loop in the new fBm algorithm is also of note. It functions

similarly to the standard octaves, but is scaled by the remainder value. It also means that the octave variable must be stored as a floating point number, rather than an integer as before. The advantage of a floating point value of octaves is that it can be generated procedurally, allowing detail to be determined by algorithmic means rather than set manually. This can be particularly useful for determining the maximum level of detail that can be displayed on screen. The author calculates that for a 2D image in screen space the value necessary to obtain the highest visible detail is:

$$octaves = log_2(screenresolution) - 2$$

In addition to this the fBm algorithm given above is then expanded to create a simple multifractal. The algorithm is very similar to the original fBm implementation but uses multiplication of the values from each octave rather than addition to create the final height value. This is referred to as a multiplicative cascade. The following is a pseudocode implementation

```
value = 1.0;

for (i = 0; i < octaves; i++)
{
     value *= ( Noise(point) + offset) * lacunarity ^ ( -H * i);
     point *= lacunarity;
}

return value;
```

Whilst looking superficially similar to the standard algorithm there are a number of changes that have been brought about. The main difference is the multiplication operation instead of an addition, but this itself brings about other changes. For instance, the value variable must now start at 1 rather than 0 and the concept of partial octaves has been removed entirely. The former is clearly necessary as 1 operates as the identity in a multiplication operation in contrast to 0 in addition operations. The latter comes about because of difficulties in managing scaling factors in multiplicative

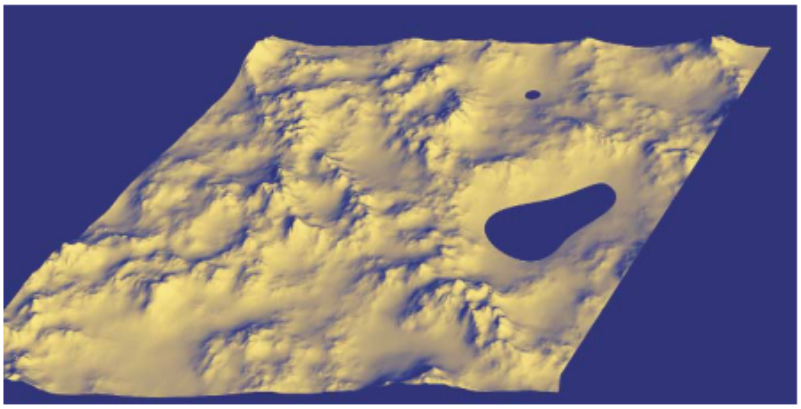

Figure 2.2: Terrain from a Multifractal. Taken from [1]

operations. Whereas in standard fBm the scaling factor operated only on one octave here such an operation would modify the entire return value.

The overall effect of the multiplicative cascade is to have each variable affect the others in a more direct way. In this case even a single octave, regardless of which one, can drastically affect the end result. The offset value here is another addition; it serves as a means of keeping some sort of coherence between the values returned by each octave. If the offset value is zero then the end result becomes wildly unpredictable and heterogeneous in the extreme. This is because each point has little reason to be related to its neighbours because of the compounding effect each minor difference has when using multiplication. This is obviously undesirable as each point needs some connection to its surrounding points in order to maintain relatively smooth and realistic terrain. The offset value aims to be the main contributing factor in the final result rather than the noise value. This allows variation from point to point but prevents each point being completely random. Obviously the value assigned to this offset is very important. If it's too high the terrain is almost entirely determined by it, and the terrain acts as a monofractal - tending towards completely planar if the value is high enough. On the other hand too low a value ensures the aforementioned problems of terrain becoming entirely heterogeneous. A value of 0.8 is suggested by the author, which is claimed to produce good results as per Figure 2.2

One of the main issues with this terrain implementation is its sheer instability. Although the offset value is intended to provide a balancing force modifying the value

assigned to it can drastically change the terrain surface. Furthermore, the offset value is generally only suitable for a set number of octaves. Changing the quantity of octaves can also change the terrain to a large degree and the ideal offset value for a given number of octaves is not defined. A final issue with this approach is that the final height once again remains completely unbounded. Unlike the additive approach though, normalising the heightmap results becomes more difficult and cannot be accomplished by a simple division.

Due to the unpredictable nature of multiplicative fractals other methods have been used to achieve more controlled results. One of these utilises a combination of multiplicative and additive techniques in an attempt to harness the best aspects of each. The overall function is additive, in that a value is computed for each octave and the combined sum of these octaves forms the final returned height value. A multiplicative aspect is introduced, however in the form of an extra weighting variable. This weighting variable is multiplied by the standard fBm value to produce a final value for each octave. The weighting value itself is determined by a multiplicative cascade of each octave result before weighting, ensuring a heterogeneous result. Any extreme divergence is kept in check, however by clamping the weighting value to values less than or equal to one.

Many fractal techniques can be created for terrain generation, each potentially producing very different results. Some approaches suggested are to modify fractal behaviour based on height, to use a power series of octaves or to include subtraction and division operations in addition to multiplication and addition. Of particular note is that one of the defining elements of terrain behaviour seems to be the underlying noise generation function. In perlin or simplex noise interpolation is achieved through a hermite spline basis function.. Changing the method of noise generation, and therefore changing the basis function, has been shown to drastically affect the final terrain. Therefore, whilst spline basis functions are suitable for reasonably rounded terrains, a different noise algorithm may produce a surface possessing different traits, even when using the same multi-fractal techniques.

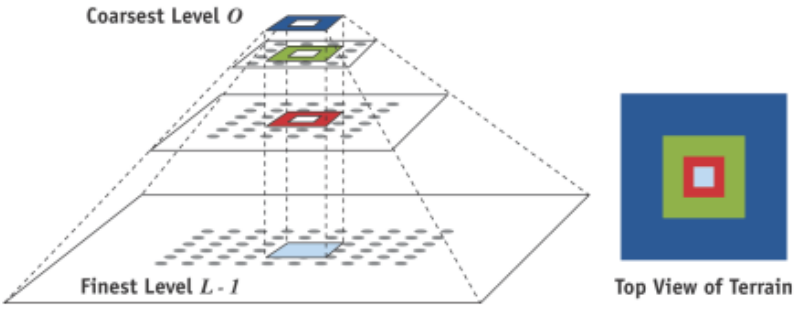

Figure 2.3: The pyramid of Clipmap layers. Taken from [4]

2.2 Geometry Clipmaps

Geometry clipmaps[3] are a LOD system used to display large amounts of terrain with a low computational cost. Terrain geometry is stored as a series of nested hierarchical grids, enabling reduction of terrain detail corresponding to distance from the camera. The highest detail terrain is focused closest to the camera. Terrain detail reduces in concentric areas spreading out from the central point, as per Figure 2.3, enabling a steady drop off in detail as the distance from the camera point to the vertices increases. As well as a decrease in mesh density the source image resolution is also decreased according to distance from the camera. The source information is therefore treated much like a mipmap, presenting lower resolution versions of a heightmap image as required. These lower resolution images are pre-filtered and stored in external memory in advance. This allows much lower use of main memory at runtime due to the fact that only a small area of the texture is stored at the highest resolution. Such a scheme can prove useful when using a heightmap of very large dimensions, e.g. tens of thousands of data points per side.

Most LOD systems for terrain suffer due to large computational and bandwidth requirements. Geometry clipmaps avoids this by using preset mesh vertices. These vertices decrease in density from the centre outwards, in a series of levels. This system avoids recalculation of vertices by moving the predefined mesh so that it is always centred around the camera position. Rather than changing the vertices themselves the heightmap data that they represent are changed as the camera moves across the terrain. This is is contrast to other LOD approaches which involve run time calculations to create and modify the vertex structure and index buffers.

The finest level of detail in both the source heightmap and the vertex mesh is represented as a square grid. Each level beyond that consists of square rings that surround the previous level. The system aims to use this series of rings to keep the screen-space distance between vertices roughly constant regardless of their distance in world space. To avoid visible differences between detail layers all vertices near the outer limits of a level are deemed to be within a transition region. The height of the vertices in this region are modified by linear interpolation between their initial position and the hypothetical position they would occupy if they were based on the next coarsest level of the heightmap. This interpolation depends on distance to the next level, so vertices far from the outside edge of level X have their height mostly determined by the level X heightmap rather than the level $X + 1$ version whilst those close to the edge have the opposite weights.

2.3 GPU-Based Geometry Clipmaps

Whilst the standard geometry clipmap method works well it is highly dependent on CPU computation to operate. As modern GPUs have advanced the bottleneck of many algorithms lies both on CPU processing speed and the speed of data transfer to the GPU. In contrast the GPU is often under-utilised therefore it is usually beneficial to transfer computational duties to it. This is not always possible, however due to the Single Instruction Multiple Data (SIMD) nature of GPU operations as well as its more limited instruction set. In their paper[4] Arul Asirvatham et al. claim to have succeeded in transferring much of the geometry clipmap operations to the GPU, achieving speeds in some cases that are an order of magnitude faster than the original implementation.

In transferring the calculations to the GPU heightmaps are stored as 2D textures. As it would be impossible to store a full detail heightmap on the GPU multiple versions are sent: one for each detail level to be used. The size of each heightmap texture also corresponds to the detail level, in that only the lowest detail heightmap texture is sent in its entirety. For each incremental detail level only the subsection that corresponds with the area that lies within its associated mesh ring is sent. This means that every mesh level will be able to use the heightmap associated with that level as well as each heightmap of a lower detail level. This attribute is important for blending calculations in transition regions as well as any fallback situations where proper detail is not able

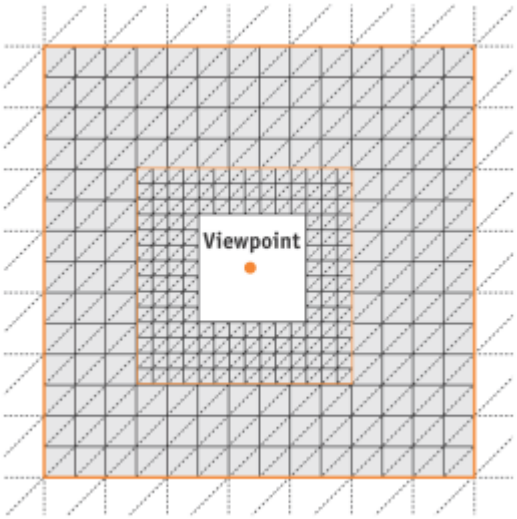

Figure 2.4: Each layer in the clipmap is imperfectly centred. Taken from [4]

to be generated within the required timespan.

Due to the semi-regular layout of the vertices in the terrain mesh each one does not have to be created and sent to the GPU. Instead, a set number of 2D grid templates are created at initialisation time and sent in a vertex buffer. From these templates the entire grid is created, by instancing the templates at various scales, translations and rotations. The distance between each vertex at a particular level is twice that of the previous level but the number of vertices remains constant. Also an odd number of vertices is chosen to enable each level to match up evenly with the next. This has the effect of ensuring that a given detail level can never be properly centred within the next coarser level (Figure 2.4), a fact that must be accounted for when using the templates to create the mesh. As hardware is often optimised for textures with sizes that are powers of two their approach uses levels with $2^n - 1$ vertices on each side, leaving the last potential row and column of the texture unused in order to keep the number of vertices odd. The authors mention a value of 255 being used predominantly in their work.

The grid templates that are sent to the GPU can create the entire mesh structure and consist of the following types:

- m x m Grid square

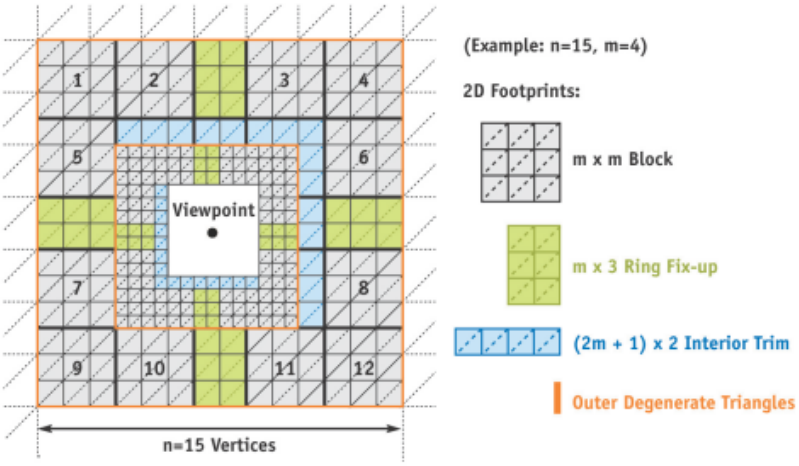

Figure 2.5: The detail layers are constructed from set templates. Taken from [4]

- m x 3 Grid rectangle
- m x 1 Interior Trim

Here the value for m is determined by

$$m = (n + 1)/4$$

and n is the number of vertices per level per side.

The manner in which these elements fit together is clear from Figure 2.5. From this it is clear that the entire grid structure can be created from this small number of template structures. Of note is an extra skirt of area zero triangles positioned between the vertices of each level. This enables a seamless mesh to be maintained even at the T-junctions between triangles of different sizes (i.e. of different levels). In the case that any point on the edge of a triangle has a different elevation than its neighbouring point in the adjacent level then the skirt will expand to link the two levels, preventing gaps in the mesh.

As the camera moves around the vertex mesh moves with it while the texture coordinates of each vertex change. A naive implementation would be to evaluate the texture coordinates from scratch each frame, but this is not strictly necessary. As most camera movement is continuous the texture coordinates will change very little from frame to frame. Therefore for each detail level most of the texture information can

be reused after a simple translation. Some new information will need to be created at the edges in the direction the camera is moving in and some is discarded at the opposite side. For this reason only a small L-shaped region of new texture information needs to be added. If the texture region for each detail level is represented with a wraparound basis then the translation can be achieved by simply moving the centre point of the texture. This means that the only an L-shaped region needs to be generated each frame, vastly reducing computational time.

It should be noted that all height information for this implementation is created in advance, rather than at runtime. This can reduce cost significantly, depending on potential runtime creation methods. It has an advantage in terms of both vertex and normal creation, as otherwise a height creation algorithm would need to be run multiple times for every vertex in the mesh: once to determine height information for that vertex and several more times to determine the normal, which is based on the height information of nearby points. Creating a heightmap in advance does have disadvantages however, in that either the heightmap image is filtered before being passed to the GPU (as is the case with this implementation) or the entire heightmap is passed to the GPU (which can be prohibitive in terms of bandwidth and memory for anything but small heightmaps). In this case the authors conducted the filtering operation on the CPU, which is relatively expensive but not so much as to cause a bottleneck. As the CPU is remaining unused for most operations in this instance this is acceptable, but if it were required to perform other tasks such as AI, physics, etc. then the extra load could become problematic.

The authors' results show that the new system is capable of rendering a heightmap of size $216,000$x$93,600$ in a 1024x768 sized window at 130 frames per second. In contrast the original approach operates on the same data set but in a 640x480 sized window at 120 frames per second. This clearly shows an improvement over the previous work but perhaps not as great a one as was expected given the speed increases in some areas of calculation. The authors acknowledge this in their work and claim the bottleneck in speed now lies in the vertex shader. The NVIDIA GeForce 6800 GT used in their test is limited in terms of its vertex texture lookup ability. As this card was one of the first to introduce HLSL shader model 3.0, and with it the very ability to perform vertex texture lookups, more recent graphics hardware should be able to remove this bottleneck. In particular HLSL shader model 4.0 introduces unified

shading architecture, allowing any type of shader computation to be performed on any of the device's many computational processors. This should result in much fewer GPU bottlenecks, as before only certain designated processing units could perform certain shader computations. For this reason it is likely that on modern graphics hardware the GPU based approach would outperform the CPU based approach to a much greater degree.

2.4 Spherical Clipmaps

In Terrain Rendering using Spherical Clipmaps[5] the authors expand the geometry clipmaps method to cover the use of spherical terrain. As geometry clipmaps cover, by default, planar terrain there are a number of approaches to extending it in order to accommodate 3D objects such as spheres [11][12][23]. Most of these approaches attempt to solve the problem of spherical terrain by taking a cube as a rough approximation of a sphere. This allows each face to be treated as a plane and calculated much as before. Problems, however, can occur at the edges between each plane. Each vertex can be modified in order for the mesh to take a spherical form yet the edges between the original planes can cause visual artifacts where they join. Special cases need to be made both for terrain near an edge and near a corner of three planes. Such an implementation can therefore become quite complex, both in terms of usability and processing time.

Clasen et al[5] attempt to alleviate these problems by designing a system based wholly on spherical terrain rather than a simple conversion of planar approaches. Whilst still utilising the general form of stacked detail levels that is demonstrated in standard geometry clipmaps, the shape of these are circular rather than square, in order to better fit the spherical structure. Therefore, instead of imagining the detail levels of the clipmap as a pyramid, as is the case with the original clipmaps, spherical clipmaps can be thought of as having a cone shaped hierarchy. The highest level is represented as a circle of vertices and each subsequent level form a concentric circular ring around the previous one as per Figure 2.6. Heightmap information is again pre-calculated, with filtered elements passed to the GPU as needed.

A major change from planar clipmaps is the parametrisation of the mesh of vertices passed to the the GPU. As it is impossible to create a truly circular form using a

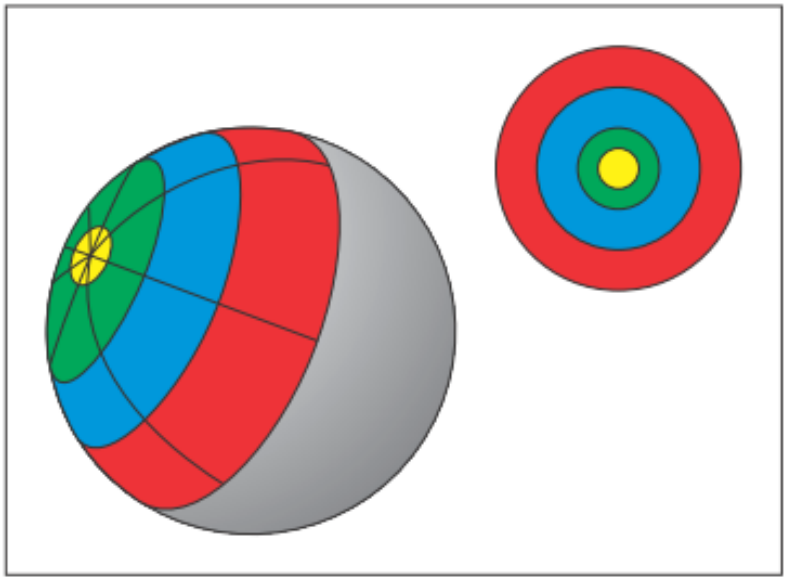

Figure 2.6: The levels in spherical clipmaps are concentric circles. Taken from [5]

standard (x, y) grid this approach must be abandoned. Instead the grid is formed from spherical coordinates. These spherical coordinates represent a hemisphere of vertices that form the clipmap mesh. A hemisphere, rather than an entire sphere, suffices in this case due to the fact that, regardless of the camera position, the maximum area of a sphere that can be seen at one time is half the surface area. As before, the hemisphere changes based on the camera, but in this case simply rotates around a fixed position to constantly face the camera position, rather than moving directly across a plane. The coordinate system of the hemisphere is based around its single pole, and the entire mesh rotates so as to place the pole at the closest vertex to the camera.

As a heightmap usually consists of a 2D plane a method of converting this 2D surface to the 3D surface of the sphere is needed. The simplest means of achieving this, and the one used by the authors, is to simply map the azimuth angle (ϕ) to the X axis of the heightmap and the inclination angles (θ) to the Y axis, ensuring that $(\phi, \theta)\epsilon[0, 2\pi)\text{x}[0, \pi)$. A point p on the surface of the sphere can therefore be represented in overall world space as $p = (x, y, z)$. It can also be represented in spherical terms in the form $p = (\phi, \theta)$. A final representation system needs to exist however to account for spherical coordinates on the rotating hemisphere. Due to the constant variation of the pole position with regard to camera location these coordinates are separate from

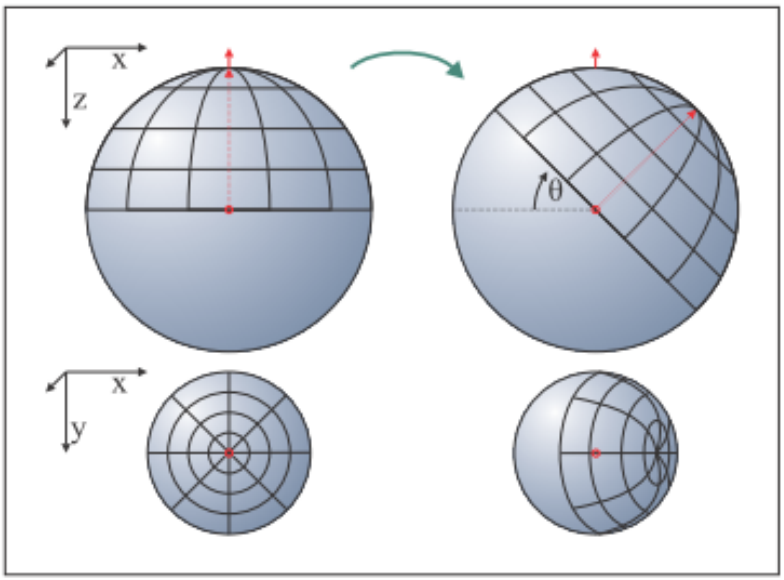

Figure 2.7: This shows the transformation of the vertex mesh. Taken from [5]

the main spherical coordinates. Therefore local coordinates on the sphere are created, represented by $p_{local} = (\phi_{local}, \theta_{local})$ where $(\phi_{local}, \theta_{local})\epsilon[0, 2PI)$ x $[0, \pi/2)$. The camera position determines the relationship between the p_{local} and p so it too needs to be represented. In this case the direction to the centre of the sphere is required, rather than the distance, so this can be set as $p_{viewer} = (\phi_{viewer}, \theta_{viewer})$. In this case the poles are defined as being on the z axis, so ϕ_{viewer} indicates rotation around the z axis and θ_{viewer} indicates rotation around the y axis as per Figure 2.7.

Each mesh vertex is represented in terms of its local hemispherical coordinates $(\phi_{local}, \theta_{local})$. In order to obtain height information for the vertex the heightmap must be queried at its associated point. This is not a trivial matter however, as the heightmap exists in world spherical coordinates (ϕ, θ). Therefore a mapping needs to be created to translate from local to world coordinates. This mapping must also incorporate the camera position so can be thought of as

$$f(\theta_{viewer}, \phi_{local}, \theta_{local}) \rightarrow (\phi, \theta)$$

The only element of the camera location that needs to be passed in this case is θ_{viewer}, due to its potential ability to modify the shape of the hemisphere grid in world

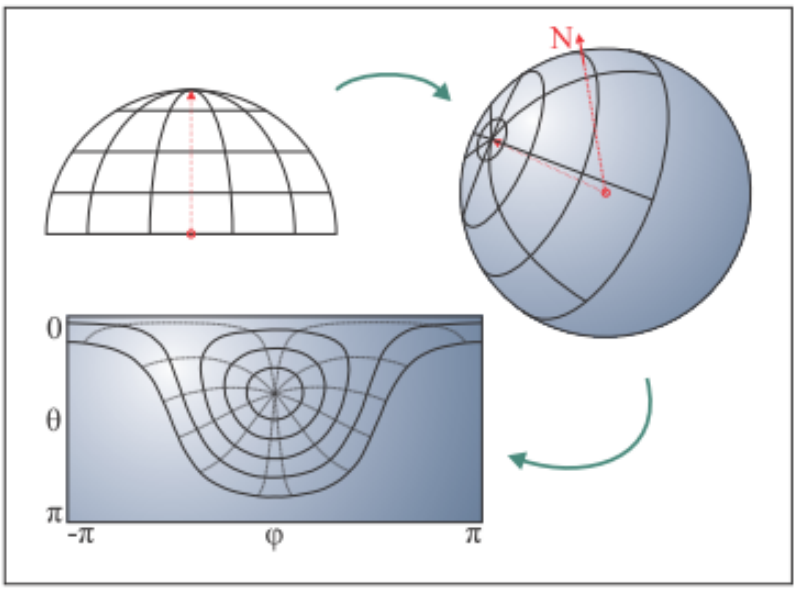

Figure 2.8: This shows the mapping operations of the mesh. Taken from [5]

coordinates. As can be seen in Figure 2.8, modifications of ϕ_{viewer} simply add a ϕ offset to the world coordinates whereas θ_{viewer} determines the boundaries of the hemisphere in this form. For this reason θ_{viewer} can be assumed to be zero and therefore removed from the above function.

Any point on the hemisphere can be found in local cartesian coordinates by

$$p_{local} = (cos(\phi_{local}).sin(\theta_{local}), sin(\phi_{local}).sin(\theta_{local}), cos(\theta_{local}))$$

To map this to world space a rotation needs to be made around the y axis, resulting in

$$p = (cos(\theta_{viewer}).p_{local}X sin(\theta_{viewer}).p_{local}Z,$$

$$p_{local}Y,$$

$$-sin(\theta_{viewer}).p_{local}X + cos(\theta_{viewer}).P_{local}Z)$$

This is converted to spherical coordinates by

$$(\phi, \theta) = (tan^{-1}(pY/pX), cos^{-1}(pZ)\theta_{viewer})$$

Subtracting θ_{viewer} from θ ensures that the local coordinate system is set to align with the viewer. Coupled with the fact that ϕ_{viewer} is defined as 0 this ensures that the local system is correctly mapped based on viewer position.

Due to the constant nature of the hemispherical grid its local coordinates will never change. For optimum speed therefore, the local cartesian coordinates of the hemisphere can be calculated at initialisation time rather than per frame. Furthermore $cos(\theta_{viewer})$ and $sin(\theta_{viewer})$ need only be calculated once per frame, so can be determined on the CPU and passed to each shader, rather than calculated per vertex per frame.

The use of a circular mesh over the standard rectangular brings about extra mapping issues but also provides many advantages in terms of providing good detail whilst minimising the number of vertices to be drawn. Representing the mesh using spherical coordinates creates an automatic LOD system due to the fact that the distance between each vertex expands with greater theta values. This uses what is generally seen as the main disadvantage of spherical coordinates: the loss of detail the further the point is from the pole, and uses it to positive effect. The pole is always the closest point to the camera, therefore should always have the highest level of detail. Detail should also decrease as distance from the pole (and therefore distance from the camera) increases.

Although the above system works well to decrease detail in the ϕ dimension it has no effect on the number of vertices used in the θ dimension. In fact decreasing resolution in one dimension while maintaining the other as static creates very long, elongated quads. This can result in visual artifacts when the heightmap is applied so a means of applying a LOD system to the θ dimension needs to be employed. This is done using the aforementioned clipmap concentric rings. Each ring implies a different level of detail in the theta dimension, increasing the distance between vertices as θ values increase. Each level I covers a θ distance given by

$$\theta\epsilon(2^{-i}\pi, 2^{-i-1}\pi]$$

with the final, highest resolution level covering

$$\theta\epsilon(2^{-i-1}\pi, 0]$$

Each level contains a constant number of vertices in the θ dimension, so as the theta distances covered by each level increase the density of vertices decreases. The number of such vertices in the θ dimension is set manually at value m. It is clear therefore that each detail level actually consists of m concentric rings, each one quad in thickness. The θ value for each ring could be obtained by linear interpolation, but in this case also increases nonlinearly to provide greater detail closer to the camera. For a given ring j, in level i with m rings per level its θ value is provided by

$$\theta_{ij} = \theta_{i0} * 2^{-j/m}$$

The overall result of these mesh modifications is that triangles take up approximately the same same screen space, regardless of the camera distance. A further major advantage is that there are no potential gaps at level boundaries therefore transition regions do not need to be used. Also, a direct $1:1$ ratio of vertex overlap between levels ensures the there are no potentially problematic T-intersections. However, the move away from rectangular grids also ensures that the original $1:1$ correspondence of vertices to height samples no longer exists, meaning that heightmap sampling will require a filtering operation.

As the shape of the extents of the hemisphere in 2D heightmap/world space can vary depending on viewer position, the size and shape of each detail level in world space will also change. If a given detail level ring contains a world-space pole then that level will contain every value of ϕ in world space. In contrast, a level that only contains coordinates near the world-space equator will only contain a small subset of all possible ϕ values. θ values will remain constant throughout.

Sampling circular sections of the heightmap at various details to perform standard clipmaps is next to impossible, due to the rectangular grid nature of the heightmap. For this reason the sampling filters used on the heightmap texture must be rectangular, and determined by the extent of their associated levels in world space. Due to the aforementioned potential variance in ϕ the filter size is not always equal in both θ

and ϕ dimensions. Further modifications are necessary to clip θ values that go beyond poles, as wraparound addressing occurs in the ϕ dimension, but not in θ (if it did the shape would be a torus).

The LOD system described up to this point will change vertex density based on the angle between the sphere and the camera position and is suitable for situations where the camera maintains a constant distance from the planet's surface. If the camera were to move away from the surface while keeping the angle constant the vertex density in world space would stay constant, meaning that in screen space it would increase with distance. Conversely, if the camera were to be positioned at a very short distance from the surface of the sphere the outer levels of the hemisphere cannot possibly be viewed by the camera, and so are drawn unnecessarily.

To combat this waste the authors also devise a means of modifying detail based on the distance to the camera by removing detail levels if they are not required. This can be achieved by finding both a maximum θ value that can be viewed from a given position as well as well as a minimum θ difference between vertices that can actually be displayed on screen. The maximum value is obtained by the formula

$$\theta_{max} = cos^{-1}(r/r + h)$$

Here r is the radius of the planet and h is the distance of the camera from the surface. It should be noted that this is only an approximation based on a smooth sphere, however. Differences in height may mean a point is sufficiently elevated to be seen when beyond this limit, so a buffer zone of some sort may be needed.

The minimum θ difference is given by

$$\theta_{diffmin} = s/r$$

where r is the radius of the planet and s is the size of the area that one screen space pixel takes up on the surface of the planet. An approximate value for s is gained from the following formula

$$s = h.tan(fov/p)$$

where h is the height of the camera above the planet's surface, fov is the field of view angle of the camera and p is the number of pixels per scanline.

Chapter 3

Design

The design of this system uses a number of the techniques already described to produce a system capable of generating and displaying multiple planetary bodies to a high level of detail at real-time rates. Heightmap information is generated on the fly for each vertex, removing the need for both precomputation and storage space for the heightmap in both system memory and GPU memory, as well as bandwidth issues from the transfer between the two mediums. As a large heightmap is necessary for extremely large-scale terrain, such as this project, these constraints can become prohibitive. By generating heightmap information as needed in real-time, computational speed is sacrificed instead of memory. Due to the ever increasing processing speed of graphics hardware this is deemed to be a good trade-off, especially as the algorithms scale very well to SIMD techniques.

A number of techniques are used to generate height information. Simplex noise is used as the base for all generating functions as it scales well to higher dimensions and offers statistically better quality noise than its earlier perlin equivalent. This implementation uses a modified version of pre-existing HLSL code [8][9][10] to generate the noise directly on the GPU. From this base of simplex noise a selection of fractal techniques are used to create a realistic terrain surface. These are:

- A standard fBm system, using the improved form as discussed previously.
- A multiplicative cascade, modified in an attempt to remove some of the unpredictability of the approach.

- A hybrid multiplicative-additive approach, using a bounded multiplicative weighting function to modify the standard additive process.
- A fractal model that generates ridged terrain. This is accomplished by allowing negative noise values as well as positive, then taking the absolute value, creating sharp ridges as well as more rounded terrain.

The mesh structure used to display the height information is heavily based on the spherical clipmaps approach. This allows a static mesh of vertices that only needs to be sent at initialisation time, rather than per frame, yet provides a robust LOD scheme specifically designed for spherical terrain. Each vertex is assigned a height based on its x,y and z coordinates, meaning that a three-dimensional heightmap needs to be generated on the fly. Although this requires some extra calculation, this scales linearly with dimension and produces very good mapping form heightmap to vertices without visual artifacts that can occur in two dimensions. These artifacts were deemed more costly and time consuming to fix than any potential speed benefits that would be obtained. Furthermore, as the entire heightmap is not generated in advance, the same number of heightmap calculations are undertaken in the three dimensional approach as in two dimensions. The generation of an entire 3D heightmap in advance would exacerbate the already expensive memory and bandwidth problems to a point where performance would likely become unacceptable. Therefore an on the fly, per vertex heightmap is currently the only way by which 3D heightmaps can be used for large amounts of high detail terrain.

This system is mostly GPU based. All generation of heightmap information, plus vertex transformation and rendering is done on the GPU. The CPU is limited to creating the vertex mesh at initialisation time, plus simple tasks such as camera control and rotation matrix creation. This design leverages the power of the graphics hardware to both generate and display large quantities of spherical terrain in real time.

Chapter 4

Implementation

4.1 Overview

This system uses DirectX 10 and HLSL 4.0 to display all terrain information on screen. To that end some generic initialisation and setup code for DirectX is taken from both the Microsoft DirectX[10] and from Luna[2]. The system enables real-time resizing of the display window and a user-controlled camera class that utilises the mouse and keyboard. Whilst a proper user interface is desirable, time constraints unfortunately made this impossible so all parameter modification is effected by modification of the source code.

The main class that controls program operation is the TerrainApp class. Much of the terrain functionality is called within the Terrain class as well as the Terrain.fx and Noise.fx HLSL effect files. A variety of other classes exist to provide various other utility functions for DirectX applications; these include classes providing camera, texture manager, skymap, lighting, and timer functionality, amongst others. While vital to program operation they are not specific to this procedural terrain application and therefore will not be described in any detail.

The TerrainApp class is a child class of the more generic d3dApp class and from it is obtained much or the functionality necessary to run a functional DirectX application. This functionality includes operating system message handling, modifiable window display and fully functional draw and update loops. A Terrain object represents an entire planet; the TerrainApp object initialises the required number of said objects along with

calling their draw and update methods. Along with the Terrain objects all other helper objects (camera, lighting, etc) are initialised and updated by the TerrainApp object. Any user input is also read in at this stage and assigned to variables in the appropriate objects. A further operation of the TerrainApp object is to analyse information received from other objects and present them on screen as statistics to the user. This includes information such as rendering time, planetary positions, viewer angles, etc.

4.2 Terrain Class

Many of the the application calculations are performed on the GPU, rather than the CPU so the role of the Terrain class is mostly limited to creation of the vertex mesh and calculating the appropriate transformation mappings based on the camera location. Upon initialisation the Terrain object is passed a number of parameters by the TerrainApp object, determining the make-up of the planet the Terrain object will represent. This information is contained within an InitInfo struct for ease of use. The struct itself is laid out as follows:

```
struct InitInfo
{
     float PlanetRadius;
     D3DXVECTOR3 PlanetPosition;
     int NumPhi;
     int NumLevels;
     int NumRingsPerLevel;
};
```

PlanetRadius: the size of the planet and therefore the size of the mesh to be constructed.
PlanetPosition: the location of the planet in world space. The mesh will rotate around this point.
NumPhi: the number of vertices used throughout in the ϕ dimension.
NumLevels: the number of detail levels to be used in the θ dimension.
NumRingsPerLevel: the number of of vertices used per detail level in the θ dimension.

Upon receiving the initialisation data the Terrain object uses it to produce the vertex and index buffers that make up the terrain mesh through the BuildVB and BuildIB methods respectively. The other principal method used in this class is the Draw method. It is here that the buffers are sent to the GPU along with a rotation matrix for the mesh. This rotation mesh is calculated in the Draw method each frame and passed to the GPU to ensure the mesh faces the camera at all times, as is required for a spherical clipmap implementation.

4.3 BuildVB method

In the BuildVB method the system must first assign memory for the vertex buffer from both the size of memory an individual terrain vertex will take up and the number of such vertices that will make up the buffer. A terrain vertex holds more than just positional data and is represented by the following struct:

```
struct TerrainVertex
{
     D3DXVECTOR3 pos;
     D3DXVECTOR2 sphericalCoord;
     int level;
};
```

pos: the 3D Euclidean position of the vertex in local space
sphericalCoord: the 2D spherical position of the vertex in local space
level: the detail level that this vertex currently lies within

From that it is clear that each TerrainVertex takes up 24 bytes on Win32 systems. In order to determine the buffer size, however, the number of these vertices needs to be established in advance. The formula is given by:

$$NumVertices = NumLevels * NumRingsPerLevel * NumPhi - (NumPhi - 1);$$

The formula is relatively straightforward, as the size of the grid can be found by multiplying the number of theta vertex location by the number of phi vertex locations. The final subtraction operation is to take into account the layout of vertices at the pole. For every other theta value there are phi vertices on that ring. However at the pole all vertices on the ring would occupy the same space therefore only one vertex is necessary.

Once the vertex buffer has been created the BuildVB() method fill it with vertex information. It loops through each value of θ and ϕ, creating a vertex at each. A pseudocode implementation of it has the following form:

```
for(currentLevel = 1; currentLevel <= numLevels; )
{
     thetaValue = getThetaValue(currentLevel, currentRing);
     for( j = 0; j < mInfo.NumPhi; ++j)
     {
          phiValue = getPhiValue(i, numPhi);
          Vector3 position =
               getSphericalCoordinates(planetRadius, thetaValue, phiValue)
          vertices[positionMarker].pos = position;
          vertices[positionMarker].sphericalCoord =
               Vector2(thetaValue, phiValue);
          vertices[positionMarker].level = currentLevel;
          positionMarker++;
     }
     currentRing++;
     if(currentRing >= numRingsPerLevel)
     {
          currentRing = currentRing % numRingsPerLevel;
          currentLevel++;
     }
}
```

In this case the counter loops through every detail level, and within it every ring, obtaining a theta value in every case. For every theta value in the mesh though there

are phi vertices, so a further loop must accommodate that., obtaining a phi value for each. Once these values have been obtained each element of the TerrainVertex struct must be assigned a value, namely the three dimensional euclidean space, the two dimensional spherical space and the current detail level. The special case of the pole vertex is accounted for manually outside of the above loop. It should be noted that level values range from 1 to numLevels whilst ring values have a range of 0 to $numRingsPerLevel - 1$. This is for computational reasons at various stages of the overall algorithm.

The getPhiValue function is relatively straightforward, as ϕ values are evenly spaced across the mesh. It only needs to be transformed to within its proper limits of $[0, 2\pi)$ by the following:

```
phi = (j / NumPhi) * (PI * 2.0f);
```

The getThetaValue function is more complex, due to the fact that the θ position varies based on detail level and ring number. It also must lie within its limits of $[0, \pi/2)$. θ values are obtained by this pseudocode:

```
levelBase = pow(2, - level * PI;
theta = levelBase * pow(2, - ring / numRingsPerLevel);
```

The levelBase variable determines the theta position of ring 0 of a given level. Each subsequent ring on that level has a θ value determined by the levelBase value. This algorithm ensures that θ distances between vertices scale continuously based on distance from the pole.

4.4 BuildIB method

Along with the vertex buffer an index buffer must also be constructed in order to define the triangle layout of the vertices. It too requires memory to be assigned in advance and is determined from the value for the total number of vertices that has been determined above. The number of triangles, or faces, to be constructed is determined by:

```
NumFaces = (NumVertices - 1) * 2 - NumPhi;
```

The minus operation exists to account for the vertex layout surrounding the pole, which consists of a ring of triangles rather than a ring of quads as is the case for every other theta value. The index buffer consists solely of integer values so occupies 4 bytes of memory.

The technique for filling the index buffer is as follows:

```
totalTheta = numLevels * numRingsPerLevel;
for (tempTheta = 1; tempTheta < totalTheta-1; ++ tempTheta)
{
     for (tempPhi = 0; tempPhi < NumPhi; ++tempPhi)
     {
          int v1 = 1 + (tempTheta-1)
               * NumPhi + tempPhi;
          int v2 = 1 + (tempTheta)
               * NumPhi + tempPhi;
          int v3 = 1 + (tempTheta)
               * NumPhi + tempPhi + 1;

          int v4 = 1 + (tempTheta-1)
               * NumPhi + tempPhi;
          int v5 = 1 + (tempTheta)
               * NumPhi + tempPhi + 1;
          int v6 = 1 + (tempTheta-1)
               * NumPhi + tempPhi + 1;

          indices[outerCount] = v1;
          indices[outerCount + 1] = v2;
          indices[outerCount + 2] = v3;

          indices[outerCount + 3] = v4;
          indices[outerCount + 4] = v5;
          indices[outerCount + 5] = v6;
```

```
            outerCount += 6;
        }

        indices[outerCount - 4] -= NumPhi;
        indices[outerCount - 2] -= NumPhi;
        indices[outerCount - 1] -= NumPhi;
}
```

This technique operates by visiting every vertex in turn and creating an associated quad at that location. The quad is constructed between the current θ value and that one below it, so the tempTheta variable must begin at 1 rather than zero. Furthermore the indices surrounding the pole are not added as they are dealt with separately. The location of the required vertex in the vertex buffer is calculated for all six vertices of the quad. These locations are then input into the index buffer to form triangles, while ensuring winding order remains constant. The loop then moves onto the next quad. The final three modifications to the buffer upon completion of the ϕ loop is important. It ensures that the quad at the end of a ring matches correctly with the starting quad. Without this change the quads would form a spiral pattern on the hemisphere rather than concentric circles, as is required.

4.5 Draw method

One of the main operations in the draw method is to calculate the correct rotation matrix to be applied to the hemisphere mesh. The original implementation calls for a complex series of mappings to be applied to a vertex whenever it is necessary to transform from one coordinate system to another. This implementation eschews that approach, abandoning a large series of calculations for each point and instead using a simple rotation matrix, requiring only a single multiplication per vertex in the GPU. The rotation matrix is created on the CPU here in the Draw method only once per frame.

To construct the rotation matrix it is necessary to have 3 vertices which represent the new coordinate system. These vertices are orthogonal and show the x, y and z-axes as they would be if the final rotation was applied to them. These are often referred

to as the right, up and direction vectors respectively. The direction vector is easy to obtain as the positions of both the sphere centre and the camera are known. Therefore the direction vector is simply the normalised difference between these two points. The right vector can be found by taking advantage of one of the limits on rotation placed on the hemisphere - the fact that it has only 2 axes of rotation. By design the hemisphere should only ever yaw and pitch, but not roll. As a result of this the right vector will always stay on the $X - Z$ plane and so can be represented by a single rotation value. This rotation value can be determined from the rotation of the direction vector when projected to the $X - Z$ plane.

This can be described in pseudo-code as :

```
Vector3 terrainToCamera =
     Vector3(GetCamera().position().x - PlanetPosition.x,
     GetCamera().position().y - PlanetPosition.y,
     GetCamera().position().z - PlanetPosition.z);



Vector3 terrainToCameraXZ =
     Vector3(terrainToCamera.x, 0.0f, terrainToCamera.z);
Normalize(terrainToCameraXZ);

yRotation = ArcCosine( DotProduct( terrainToCameraXZ, origTarget) );
if (GetCamera().position().x < PlanetPosition.x)
     yRotation *= -1;
```

To use these rotation values an original set of right, up and direction is created at initialisation time and maintained as a constant reference. This is what is being referred to in the origTarget vector above. New vector directions are then obtained by rotation of these reference values by the calculated rotation value. The if statement above is to ensure rotation stays in range of $-\pi$ to $+\pi$. Once the direction and right vectors have been found the up vector can be obtained by a simple cross product operation as seen here:

```
Matrix yRotationMatrix;
```

```
yRotationMatrix = CreateRotationAroundY( yRotation);
Vector3 rotatedRight;
rotatedRight = TransformVector(origRight, yRotationMatrix);
Normalize(rotatedRight);
Normalize(terrainToCamera);

Vector3 rotatedUp;
rotatedUp = CrossProduct(rotatedRight, terrainToCamera);
Normalize(rotatedUp);
```

Now that the three vectors have been obtained a rotation matrix can be formed. This rotation matrix can be combined with the translation matrix in order to modify the planet's position as well as its rotation to form a world matrix. This takes the following form:

```
Matrix terrainWorld = Matrix(
     rotatedRight.x, rotatedRight.y, rotatedRight.z, 0.0f,
     rotatedUp.x, rotatedUp.y, rotatedUp.z, 0.0f,
     terrainToCamera.x, terrainToCamera.y, terrainToCamera.z, 0.0f,
     PlanetPosition.x, PlanetPosition.y, PlanetPosition.z, 1.0f);
```

Once the world matrix is created it is passed, along with the vertex and index buffers, to the GPU where height information is created and applied

4.6 Terrain file

Information is passed from CPU to GPU in the form of vertex and index buffers, as well as separate buffers containing other variables that are needed in the GPU based algorithms. As bandwidth between these devices can be an issue most programs attempt to limit the amount of information passed between the two. This program passes little extra information so this should not be an issue, however it is still advisable to create buffers efficiently to avoid unnecessary bandwidth use. This system uses two different buffers for passing information; one for values that remain static throughout the program's operation and another for values that may change from frame to frame.

Because of their mode of operation of buffers, if one value needs to be updated then the entire buffer is updated. Grouping of variables based on their update rates reduces the bandwidth used by the application by removing many potentially unnecessary updates. The buffers used in this file consist of

Information is passed from CPU to GPU in the form of vertex and index buffers, as well as separate buffers containing other variables that are needed in the GPU based algorithms. As bandwidth between these devices can be an issue most programs attempt to limit the amount of information passed between the two. This program passes little extra information so this should not be an issue, however it is still advisable to create buffers efficiently to avoid unnecessary bandwidth use. This system uses two different buffers for passing information; one for values that remain static throughout the program's operation and another for values that may change from frame to frame. Because of their mode of operation of buffers, if one value needs to be updated then the entire buffer is updated. Grouping of variables based on their update rates reduces the bandwidth used by the application by removing many potentially unnecessary updates. The buffers used in this file consist of

```
cbuffer cbPerFrame
{
float4x4 gWorld;
float4x4 gWVP;
float4 gDirToSunW;
};

cbuffer cbFixed
{
float planetRadius;
int maxLevels;
};
```

The planet radius and the max number of levels are set at initialisation time and therefore will never be changed. The direction to sun variable could potentially be moved to the fixed buffer, as in this implementation it is never changed. Future iterations of the program, however, will likely require dynamic lighting, therefore it is left

in the per-frame update buffer for now.

The vertex shader receives as input the information contained in a TerrainVertex struct described previously, consisting of euclidean coordinates, spherical coordinates and the detail level in which the vertex lies. Vertex shader operation proceeds as follows:

Firstly the euclidean coordinates are transformed from local coordinates to world coordinates through matrix multiplication with the world matrix. This provides the translation and rotation operations needed to place the vertex correctly and enables use of these coordinates as parameters to the noise function. Using local coordinates would not have the desired effect as these are static for each coordinate, which would result in terrain that doesn't change yet always faces the camera. Secondly, a height value for these coordinates is obtained by calling upon one of the fractal generating functions. These take in the world coordinates plus a number of configuration parameters and return a single height value. Finally the height value is transformed to a vector normal to the surface of the sphere at that point and added to the world coordinates to obtain a final vertex position.

This is represented in the vertex shader code as:

```
float4 posW = mul(vIn.posL, gWorld);

float heightToAdd = fBm(posW / 10.0f, 8, 2, 1.0f);
float3 tempVector = vIn.posL;
float3 direction = normalize(tempVector);
tempVector = direction * (planetRadius + heightToAdd * 2.0f);

vOut.posH = mul(float4(tempVector, 1.0f), gWVP);
```

posW: the position in terms of world coordinates
posL: the position in terms of local coordinates
gWorld: the world matrix to be applied
fBm: the height generating method (in this case fractional brownian motion)
posH: the final vertex position after height modifications have been applied
Note that some scaling modifications have been made at certain stages. These are simply tweaking modifications to ensure a more visually appealing result and are not

strictly necessary. Other additions include a rudimentary lighting system which provides a shading value to each vertex based on simple spherical normal values.

The pixel shaders do very little in the current implementation; they simply colour the pixel based on height information, then scale this colour based on the shading factor. If textures or other more elaborate lighting features were used this is where they would be implemented. A per-vertex lighting system was created and still exists in the code but is not currently activated as it incurs a large performance penalty.

4.7 Noise file

The noise file contains a library of noise and fractal creation methods that are called upon in the vertex shaders. The noise algorithm used in this case is a HLSL version of Simplex Noise, based on pre-existing implementations[8]. Unlike Perlin Noise it does not generate noise values from Hermite spline interpolation of surrounding points but rather uses a simple additive contribution from each, with a weighting factor based on distance. It first skews the input coordinates to as to be accessible in simplex space then finds the surrounding simplex (in this case a tetrahedron) which contains the point. Gradient values are then obtained each point of the tetrahedron and these are used to interpolate a value for the point in question. The code for this method is relatively large so is not included here but can be found on the accompanying CD of source code.

Most GPU interpretations of the Simplex Noise algorithm do not offer truly random noise, as random number generation is difficult to achieve on the GPU. Instead they hard-code a pseudo-random series of numbers and store these as constants in graphics memory, providing only a limited degree of randomness. As all noise functions utilise random number functions as input this limitation can seriously affect the final terrain. This implementation circumvents this limitation by generating random numbers on the CPU, storing them as a one-dimensional texture and then passing them to the GPU. This texture is created at initialisation time and passed only once to the graphics hardware. Continuous generation of random numbers is not required because the heightmap is constant, therefore requires a constant set of numbers to achieve the same results each frame.

The other methods provided in the Noise file consist of a variety of heightmap

generating techniques using fractal methods. These are based on implementations described previously and consist of:

- fBm
- MultiCascade
- HybridMultiFractal
- RidgedMultiFractal

4.8 fBm

This is the simplest form of fractal generation and has already been discussed thoroughly. It produces a relatively homogeneous heightfield. The code consists of

```
float fBm( float4 vInputCoords,
           uniform uint nNumOctaves,
           uniform float fLacunarity,
           uniform float HValue )
{
    float fNoiseSum = 0;
    float fAmplitude;

    float4 vSampleCoords = vInputCoords;

    for( uint i=0; i<nNumOctaves; i++ )
    {
        fAmplitude = pow( fLacunarity, -HValue* i);
        fNoiseSum += fAmplitude *
             SimplexNoise3D( vSampleCoords );

        vSampleCoords *= fLacunarity;
    }
```

```
    return fNoiseSum;
}
```

In this case, along with the input coordinates, parameters for the number of octaves, the lacunarity and the H value also need to be provided. It loops through each octave, calling upon the SimplexNoise3D function each time. It can therefore be clearly seen that the computational cost of the overall method is based mainly on the number of octaves used. Of note is that the overall height is unbounded, as the division of accumulated amplitude values has not been included. This is not seen as a problem as the purely additive function is well bounded.

4.9 MultiCascade

This is a highly unstable fractal generating procedure that produces very heterogeneous heightfields. It operates similarly to fBm, but uses a multiplicative cascade rather than an additive cascade.

```
float MultiCascade( float4 vInputCoords,
     uniform uint nNumOctaves,
     uniform float fLacunarity,
     uniform float HValue )
{
    float fNoiseSum = 1.0f;
    float fAmplitude;
    float fAmplitudeSum = 1.0f;
    float offset = 0.8f;

    float4 vSampleCoords = vInputCoords;

    for( uint i=0; i<nNumOctaves; i++ )
    {
        fAmplitude = pow( fLacunarity, -HValue* i);

        fNoiseSum *= fAmplitude *
```

```
            (SimplexNoise3D( vSampleCoords ) + offset);

        fAmplitudeSum *= fAmplitude;
        vSampleCoords *= fLacunarity;
    }

    fNoiseSum /= fAmplitudeSum;

    return fNoiseSum;
}
```

The parameters are kept identical to the fBm function, but an internal offset variable is used to control the terrain and remove any overly unstable effects. An extra addition to the approach is a division operation designed to limit the potential values returned by the process. This division removes the multiplicative effects on the amplitude elements but maintains its affects on the direct noise values.

4.10 HybridMultiFractal

This process combines the additive cascade of the fBm process with the multiplicative cascade of the MultiCascade process. The result is an attempt to maintain the heterogeneous nature of the latter's heightmaps but with a level of control similar to the former. It also uses a static array of weights to modify the noise values rather than generating them repeatedly for each vertex. The first vertex to run this code initialises the exponent array, setting weight values for each octave that will be used for each subsequent vertex. The code consists of the following:

```
float HybridMultiFractal( float4 vInputCoords,
     uniform uint nNumOctaves,
     uniform float fLacunarity,
     uniform float HValue)
{
     float offset = 0.7f;
     float frequency, result, signal, weight;
```

```
    static bool first = true;
    static float exponentArray[100];

    // set exponent array
    if (first)
    {
         frequency = 1.0;
         for (int i=0; i<=nNumOctaves; i++)
         {
              exponentArray[i] = pow( frequency, -HValue);
              frequency *= fLacunarity;
         }
    first = false;
    }

    // get first octave
    float noise = SimplexNoise3D(vInputCoords);
    result = (noise + offset) * exponentArray[0];
    weight = result;
    vInputCoords *= fLacunarity;

    for (int i=1; i<nNumOctaves; i++)
    {
         weight = saturate(weight);
         noise = SimplexNoise3D(vInputCoords);
         signal = ( noise + offset ) * exponentArray[i];
         result += weight * signal;
         weight *= signal;
         vInputCoords *= fLacunarity;
    }

    return result;
}
```

A further weight value is obtained from a multiplicative cascade of height values from each octave up to the current. The height information from this octave is scaled by this weight value but the overall result is simply added to the total, rather than multiplied.

4.11 RidgedMultiFractal

This process uses a development of the hybrid approach discussed above to create heightmaps with sharp edges as well as rounded features. In this case the noise value is scaled to have a range of -1 to +1, then the absolute values of this noise are found and used instead. The advantage of this is that the surface becomes discontinuous at noise values of zero, producing a sharp ridge. This value is then squared to produce an even more variable effect.

```
float RidgedMultiFractal(
     float4 vInputCoords,
     uniform uint nNumOctaves,
     uniform float fLacunarity,
     uniform float HValue)
{
     float offset = 1.0f;
     float gain = 2.0f;
     float result, frequency, signal, weight;
     static bool first = true;
     static float exponentArray[9]; // Blatant hack. nNumOctaves+1

     if (first)
     {
          frequency = 1.0f;
          for (int i = 0; i<=nNumOctaves; i++)
          {
               exponentArray[i] = pow(frequency, -HValue);
               frequency *= fLacunarity;
```

```
    }
    first = false;
}

// get first octave
signal = SimplexNoise3D(vInputCoords);

// get signal value between -1 and +1
signal = signal * 2.0f - 1.0f;

// get ridges through finding absolute value
signal = abs(signal);
signal = offset - signal;
// square it to increase jaggedness
signal *= signal;

result = signal;
weight = 1.0f;

for (int i = 1; i < nNumOctaves; i++)
{
    vInputCoords *= fLacunarity;

    weight = signal * gain;
    weight = saturate(weight);

    signal = SimplexNoise3D(vInputCoords);
    signal = abs(signal);
    signal = offset - signal;
    signal *= signal;

    signal *= weight;
    result += signal * exponentArray[i];
```

```
        }

        return result;
    }
```

Chapter 5

Evaluation

This system was tested on a computer with the following specifications:

- Windows 7 Professional 64 bit (program running in 32 bit mode)
- Pentium Q6600 Quad-core 2.4 GHz processor
- Radeon HD4870 1GB graphics device
- 3GB 800MHz DDR2 RAM

All screenshots provided display the height-adjusted terrain without realistic lighting or textures. To enable the viewer to view the characteristics of the terrain clearly a shading factor is applied based on the height of the terrain. Large height values are shown as bright terrain and smaller height values are shown as dark terrain.

5.1 Results

Overall the results of this project are very positive. The main goal of creating systems of planets whose terrain is generated entirely procedurally was attained. Furthermore this system is capable of creating a variety of different terrain types that show variance over a number of scales. The performance of the system is also very satisfactory, being able to display an earth-sized planet (radius 6,378,100 metres) up to a definition of 0.2 metres per vertex while maintaining a frame rate of 85 FPS. A number of large planets

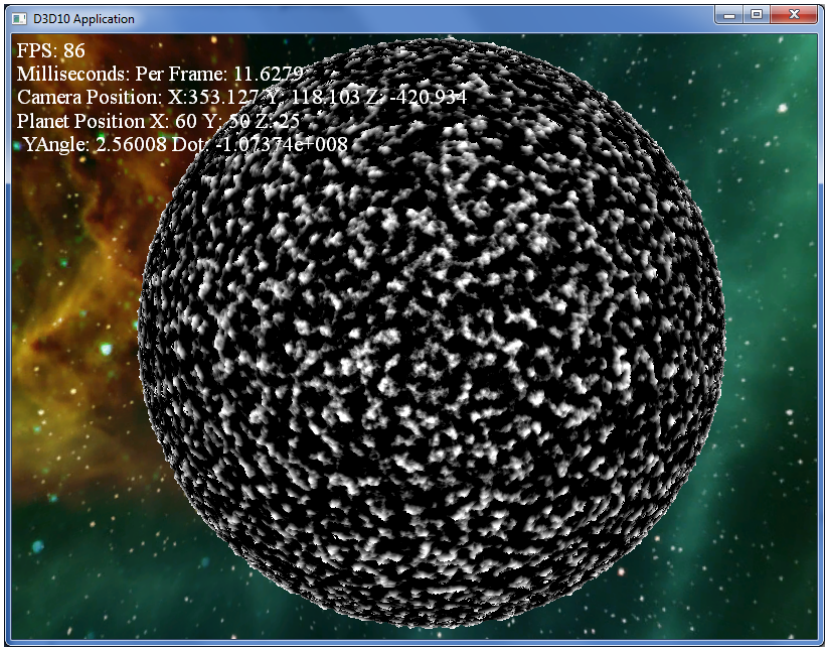

Figure 5.1: This shows the effect of fBm

can be created and displayed simultaneously while still maintaining interactive frame rates.

The fractal methods used proved effective in the creation of terrain that is subjectively both realistic and varied. However, the variety or parameters used can mean that even a well performing algorithm can produce poor quality terrain if these parameters are set to unsuitable values. Unfortunately the assigning of these values often requires much trial and error before satisfactory results can be obtained. The problem is compounded by the fact that ideal values vary from technique to technique, with the result that using constant values for all techniques results in overall poor performance.

With suitable parameters in place, however, quite positive results can be obtained. Although fBm is the simplest technique, it can still produce effective, if repetitive terrain, a can be seen in Figure 5.1

The MultiCascade technique is perhaps the hardest to obtain proper parameters for due to its instability. In Figure 5.2 this characteristic is demonstrated. While overall the terrain is visually appealing there are a number of extreme spikes in the terrain that are difficult to remove entirely. These hamper the believability of the final terrain.

The HybridMultiFractal (Figure 5.3) technique is a better alternative to the stan-

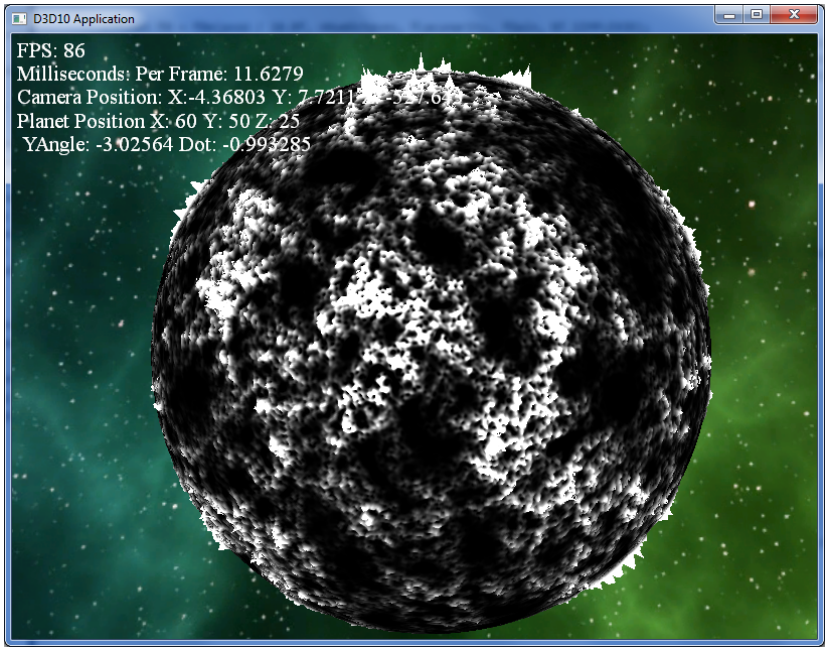

Figure 5.2: This shows the effect of a multiplicative cascade

dard fBm technique, producing more varied terrain without the extremes that the MultiCascade can generate. Unfortunately it does not maintain all of the heterogeneity of this approach and the overall result is that of rolling hills interspersed with some flatter valley areas.

The RidgedMultiFractal technique succeeds in producing terrain that is noticeably different from the other techniques, as can be seen in Figure 5.4. Of note are the distinctive lines of sharp terrain (in this case taking the form of inverted ridges) that wander across the surface. Although perhaps not realistic in its shape, this method certainly produces interesting terrain, appearing in some areas akin to noise produced from entirely different techniques, such as Voronoi noise.

The lighting system used in this project is rather rudimentary, consisting of per-vertex lighting over the surface of the planet but using a simple sphere as an approximation of the planet for the evaluation of vertex normals. A more accurate approach, using normals obtained from the surface after heightmaps have been applied, is possible and was tested in the course of development. It was found to have a hugely detrimental effect on performance, however as it required at the very least three height calculations for each vertex (in order to find the heights of neighbouring vertices) rather than one.

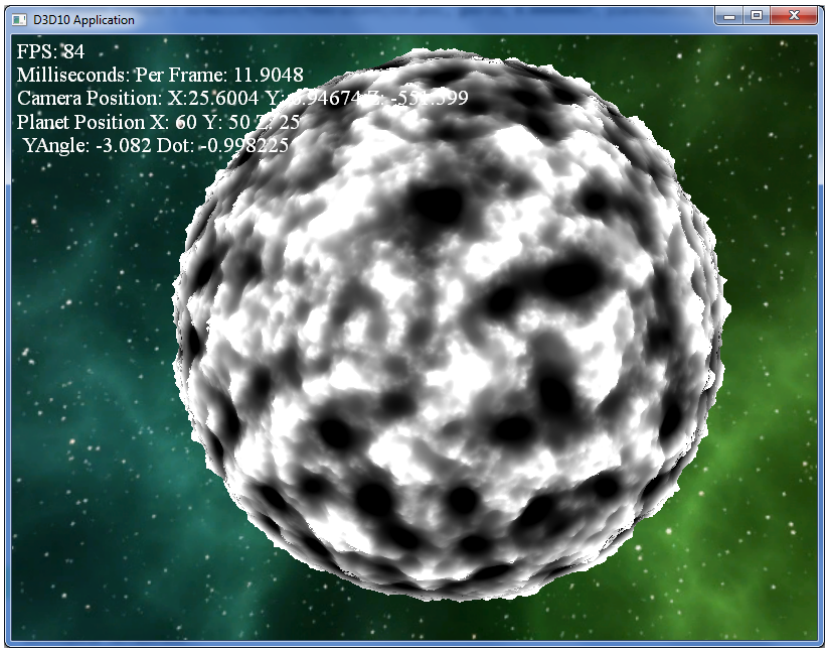

Figure 5.3: This shows the effect of the hybrid multifractal

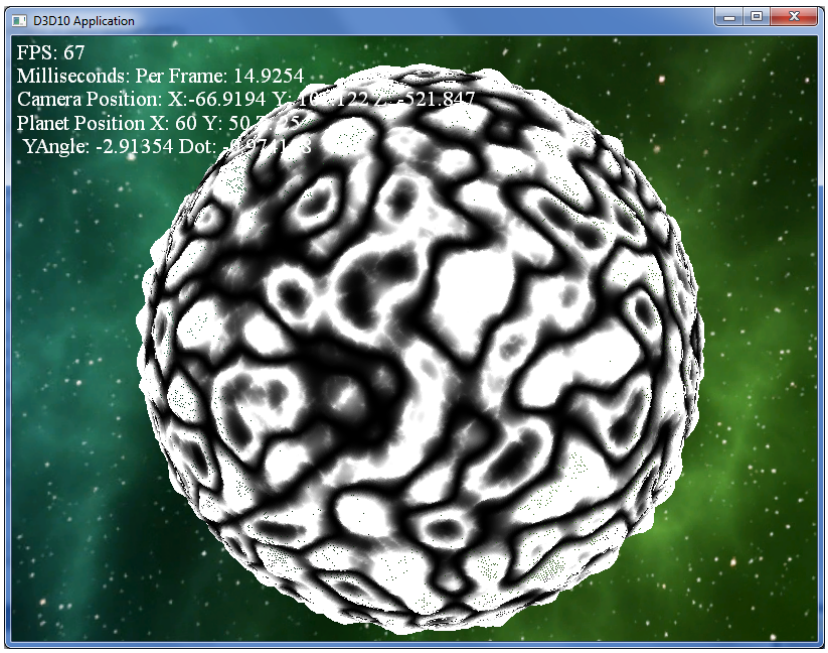

Figure 5.4: This shows the effect of the ridged multifractal

The generation of heightmap information appears to be the bottleneck in the current implementation so this causes large slowdowns. Although usable on smaller terrain it is felt that this is too much of a loss in performance to justify keeping in its current form as it is not one of the primary focuses of the project.

Another deficiency lies in the spherical clipmap LOD system used to display terrain. Although this functions well in some regards, adjusting automatically to camera position, it fails to take into account the distance of the camera from the terrain, thereby potentially drawing unnecessarily high levels of detail in some circumstances. Although an algorithm for reducing mesh complexity based on height exists, by definition this would require the mesh to be recalculated, removing the advantage of maintaining a constant vertex mesh. Generating the mesh per frame rather than at initialisation time, as this approach would seem to imply, proves prohibitively expensive without substantial modification.

5.2 Potential Improvements

Although this system performs well there are numerous improvements that could be made to bring about better results. The most obvious is to improve the lighting and the LOD system to remove the issues mentioned above. Possible solutions to the first include evaluating normals at a much lower resolution than vertices, or improving the speed of the fractal generating algorithms. The latter could be solved by removing the necessary vertices from the existing mesh rather than simply creating a new one. This, of course would also require modification of the index buffer. Another solution might be to use geometry shaders to cull certain vertices on the GPU rather than the CPU.

Numerous cosmetic improvements could be made to the terrain itself, such as the use of textures for example. A more interactive user interface would prove both practical and appealing, particularly the use of sliders to control fractal parameters in real time. This would allow the user to see the results of each parameter change instantaneously, and lead to a greater understanding of their function.

Finally, a greater variety of fractal techniques could be explored. Although this project uses a number of techniques an almost infinite variety of possible terrains could be created from fractal-based operations. Only simple operations of addition and multiplication have been employed in this project so far. More advanced techniques

have the potential to create much more varied and interesting landscapes.

Chapter 6

Conclusions

This dissertation investigated the use of procedural generation to create numerous three-dimensional planetary bodies, each displaying varying and realistic terrain. A number of noise-based techniques were used to create heightmaps - representing the terrain topography. The effects of using different techniques to generate the final heightmap surface were investigated and it was found that substantial variation could arise from minor algorithmic changes. The parametrisation of these techniques was also explored, showing that minor changes to input parameters had a similar and unpredictable effect. Despite these difficulties the project was successful in producing varied and interesting terrain, which possessed detail and displayed heterogeneity at a number of scales.

To display large amounts of terrain to the user a number of steps had to be taken to reduce the computation required and bring about acceptable real-time performance. To accomplish this many of the algorithms were designed to utilise the extra processing power of the graphics hardware, rather than the CPU. An LOD system using spherical clipmaps was implemented to reduce the quantity of vertices necessary to maintain a suitable quality representation of the terrain. An efficient noise algorithm was created on the GPU that allowed heightmaps to be generated on the fly rather than in advance. Generating such information every frame brought about substantial savings in memory and bandwidth, but with increased computational cost.

Nevertheless the system was successful in creating and displaying extremely large quantities of high-quality terrain to the user. For an earth-sized planet the methods

employed here enabled a level of detail to be displayed of as little as 20cm per vertex, while generating topographical information for the entire planet every frame, and maintaining a frame rate of 85 FPS. Further additions proposed can both improve this performance and bring about more realistic and customisable terrain.

Bibliography

[1] Texturing and Modeling - A Procedural Approach;
David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, Steven Worley;
Morgan Kaufman Publishers; 2002

[2] Introduction to 3D Game Programming with DirectX 10;
Frank Luna;
Wordware Press;
2008

[3] Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids;
Frank Losasso, Hugues Hoppe;
Stanford and Microsoft Research;
2004

[4] Terrain Rendering Using GPU-Based Geometry Clipmaps;
GPU Gems 2;
Arul Asirvatham, Hugues Hoppe;
Addison Wesley;
2005

[5] Terrain Rendering using Spherical Clipmaps;
Malte Clasen and Hans-Christian Hege;
Eurographics/ IEEE-VGTC Symposium on Visualization;
2006

[6] Presentation on Perlin Noise;
Ken Perlin;
http://www.noisemachine.com/talk1/;
Retrieved September 10 2010

[7] Noise Hardware;
Ken Perlin;
http://www.csee.umbc.edu/ olano/s2002c36/ch02.pdf;
Retrieved September 10 2010

[8] Simplex Noise demystified;
Stefan Gustavson;
http://www.csee.umbc.edu/ olano/s2002c36/ch02.pdf;
Retrieved September 10 2010

[9] Perlin Fire;
Andrei Tatarinov;
http://developer.download.nvidia.com/SDK/10/direct3d/Source/PerlinFire/doc/PerlinFire.pdf;
Retrieved September 10 2010

[10] Microsoft DirectX 10 SDK June 2010;
http://msdn.microsoft.com/en-us/directx/default.aspx;
Retrieved September 10 2010

[11] ROAM - Real-time Optimally Adapting Meshes;
Mark Duchaineau et al;
IEEE Visualization Proceedings;
1997

[12] Geomipmapping - Fast Terrain Rendering using Geometrical Mipmapping;
WH de Boer;
flipCode featured articles,

October 2000

[13] NASA Visible Earth;
http://visibleearth.nasa.gov;
Retrieved September 10 2010

[14] ImageMagick;
www.imagemagick.org;
Retrieved September 10 2010

[15] Terragen;
http://www.planetside.co.uk/;
Retrieved September 10 2010

[16] Bryce;
http://www.daz3d.com/i.x/software/bryce/-/;
Retrieved September 10 2010

[17] World Machine;
http://www.world-machine.com/index.php;
Retrieved September 10 2010

[18] Inception Visual Effects;
http://www.wired.com/underwire/2010/07/inception-visual-effects;
Retrieved September 10 2010

[19] Virtual Airspace Simulation Technologies Real Time;
http://www.aviationsystemsdivision.arc.nasa.gov/facilities/vast/index.shtml;
Retrieved September 10 2010

[20] Advanced drilling simulators offer realistic models to reduce crews learning curve;
http://www.drillingsystems.com/files/newsitem$_2$9$_I$$ADC_D rilling_C ontractor_J uly_0$6$-Simulators.p$
$RetrievedSeptember$102010

[21]$EagleLanderSimulation$;
$http://eaglelander3d.com/$;
$RetrievedSeptember102010$

[22]$MarsSimulationProject$;
$http://mars-sim.sourceforge.net/$;
$RetrievedSeptember102010$

[23]$AReal-TimeProceduralUniverse$;
$SeanO'Neil$;
$http://www.gamasutra.com/view/feature/3098/a_realtime_procedural_universe.php$;
$RetrievedSeptember102010$

[24]$RealtimeProceduralTerrainGeneration$;
$JacobOlsen$;
$http://oddlabs.com/download/terrain_generation.pdf$;
$RetrievedSeptember102010$

[25]$GrandTheftAutoSeries$;
$RockstarGames$;
$1997-2009$

[26]$RedFactionGuerilla$;
$VolitionGames$;
2009

[27]$Fallout3$;
$BethesdaSoftworks$;
2006

[28]$Interview:CodiesonFUEL$;
$JimRossignol$;

$http://www.rockpapershotgun.com/2009/02/24/interview-codies-on-fuel/$;
$Retrieved2010-03-06$.

[29]$Infinity: TheQuestforEarth$;
$F.Brebion$;
$http://www.infinity-universe.com/Infinity/$;
$RetrievedSeptember102010$

[30]$APrototypeofMarineSearchandRescueSimulator$;
$LiuXiuwen, XiaoFangbing, JinYicheng$;
$InternationalConderenceonInformationTechnologyandComputerScience$;
2009

[31]$Multiresolutionmodelsfortopographicsurfacedescription$;
$L.DeFloriani, P.Marzano, andE.Puppo$;
$TheVisualComputer$, 12(7) : 317 − −345, $August1996$.